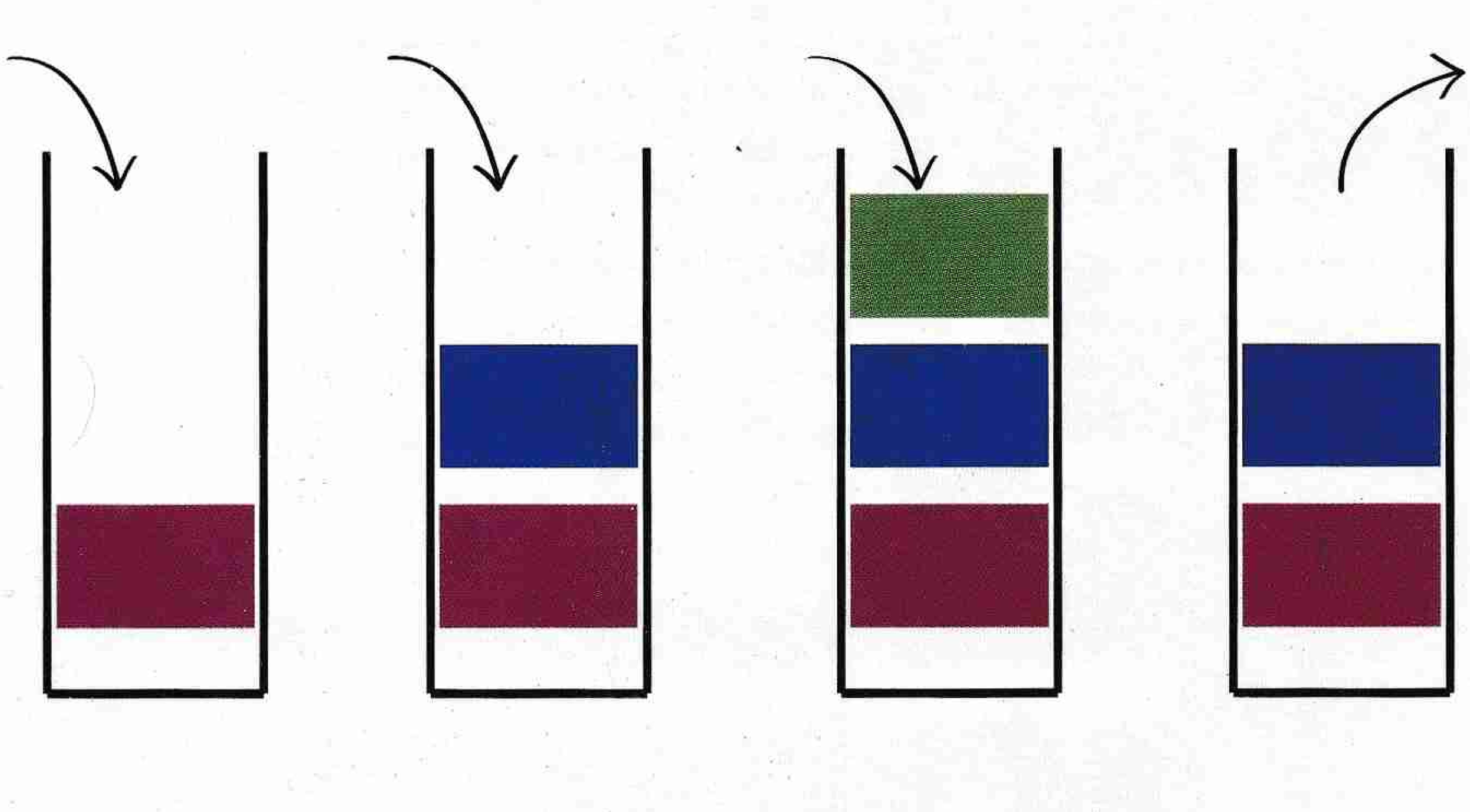
কম্পিউটার প্রোগ্রামিং

ডেটা স্ট্রাকচার ও অ্যালগরিদম পরিচিতি

তৃতীয় খণ্ড

তামিম শাহ্‌রিয়ার সুবিন



দ্বিমিক প্রকাশনী

কম্পিউটার প্রোগ্রামিং

৩য় খণ্ড

ডেটা স্ট্রাকচার ও অ্যালগরিদম পরিচিতি

তামিম শাহ্‌রিয়ার সুবিন

দ্বিমিক প্রকাশনী

কম্পিউটার প্রোগ্রামিং ৩য় খণ্ড

ডেটা স্ট্রাকচার ও অ্যালগরিদম পরিচিতি

তামিম শাহ্‌রিয়ার সুবিন

প্রথম প্রকাশ : ফেব্রুয়ারি ২০১৮

প্রকাশক :
দ্বিমিক প্রকাশনী, কনকর্ড এম্পোরিয়াম, ২৫৩-২৫৪ এলিফ্যান্ট রোড, কাঁটাবন, ঢাকা।
ওয়েবসাইট: www.dimik.pub; ফোন : 017 0796 0372
facebook.com/DimikProkashoni/

মুদ্রণ :
আলফা প্রিন্টার্স
১৬৭ গাউসুল আজম মার্কেট, নীলক্ষেত, ঢাকা

প্রাপ্তিস্থান :
দ্বিমিক প্রকাশনী, কনকর্ড এম্পোরিয়াম, ২৫৩-২৫৪ এলিফ্যান্ট রোড, কাঁটাবন, ঢাকা।
ওয়েবসাইট: www.dimik.pub; ফোন : 017 0796 0372

মূল্য : ২৫০ টাকা

Computer Programming Part 3
Introduction to Data Structure & Algorithm
By Tamim Shahriar Subeen
First Published: February 2018

Price: 250 Taka
ISBN: 978-984-34-3694-8

কম্পিউটার প্রোগ্রামিং

৩য় খণ্ড

ডেটা স্ট্রাকচার ও অ্যালগরিদম পরিচিতি

তামিম শাহ্‌রিয়ার সুবিন

দ্বিমিক প্রকাশনী

সূচিপত্র

ভূমিকা

ভালো প্রোগ্রামার হতে গেলে দরকার হচ্ছে অনুসন্ধিৎসু মন। এটি অবশ্য কেবল প্রোগ্রামিংয়ের ক্ষেত্রেই নয়, বরং আরো অনেক ক্ষেত্রেই দরকার। বাংলাদেশে থাকার সময় যখন গণিত অলিম্পিয়াডের স্বেচ্ছাসেবক হিসেবে বিভিন্ন জায়গায় ঘুরে বেড়াতাম, সেখানে স্কুল-কলেজপড়ুয়া ছেলেমেয়েরা আমার কাছে প্রোগ্রামিং নিয়ে জানতে চাইত। তাদের সেই জানার আগ্রহ আমাকে মুগ্ধ করলেও তারা ঠিক কোন বই পড়ে প্রোগ্রামিং শুরু করবে, সেই প্রশ্নের সঠিক উত্তর আমার জানা ছিল না, কারণ আমাদের দেশের স্কুলের শিক্ষার্থীদের পক্ষে একটি ইংরেজি বই দিয়ে প্রোগ্রামিং শেখাটা বেশ কঠিনই বটে। তাই তাদের জন্য ২০০৯ সালে একটি বই লেখা শুরু করি এবং ২০১১ সালে সেটি প্রকাশিত হয়, কম্পিউটার প্রোগ্রামিং নামে। তার কয়েক মাস পরে http://cpbook.subeen.com ওয়েবসাইটে আমি বইটি সবার জন্য উন্মুক্ত করে দিই। কিন্তু শিক্ষার্থীদের জানার ও শেখার আগ্রহ যেন আরো বেড়ে গেল। প্রোগ্রামিংয়ের বেসিক তো শিখলাম, কিন্তু আরো ভালোভাবে সি শিখব কীভাবে? এই প্রশ্নের উত্তর দেওয়ার জন্য লিখে ফেললাম, কম্পিউটার প্রোগ্রামিং ২য় খণ্ড। বইটি ২০১৬ সালে প্রকাশিত হয়।

এর মধ্যে বাংলাদেশের স্কুল-কলেজের শিক্ষার্থীদের মধ্যে প্রোগ্রামিং ধীরে ধীরে জনপ্রিয় হয়ে উঠছে। তারা বিভিন্ন প্রোগ্রামিং প্রতিযোগিতায় অংশগ্রহণ করে। প্রোগ্রামিং প্রতিযোগিতার অনেক সমস্যার সমাধান করতেই ডেটা স্ট্রাকচার ও অ্যালগরিদম বিষয়টি জানা প্রয়োজন। তাই আমার কম্পিউটার প্রোগ্রামিং ১ম খণ্ড ও ২য় খণ্ড বইয়ের পাঠকদের ডেটা স্ট্রাকচার ও অ্যালগরিদম-এর সঙ্গে পরিচয় করিয়ে দেওয়ার জন্যই এই বইটি লিখলাম। আশা করি, বইটি পড়লে প্রোগ্রামিংয়ের সৌন্দর্য শিক্ষার্থীদের মধ্যে নতুনভাবে ধরা দেবে, তাদের মধ্যে যৌক্তিক চিন্তাভাবনার ভিত গড়ে উঠবে এবং তারা একটু অন্যভাবে ভাবতে শিখবে। আমি যেহেতু বইটি স্কুল-কলেজের শিক্ষার্থীদের জন্য লিখেছি, তাই বইটি যারা পড়বে, তারা যে সবাই ভবিষ্যতে কম্পিউটার বিজ্ঞান বিষয়ে পড়বে এমনটি আমি আশা করি না, চাইও না। বিভিন্ন শিক্ষার্থী বিভিন্ন বিষয়ে উচ্চশিক্ষা গ্রহণ করবে, তাদের নিজেদের পছন্দ বিবেচনা করে ও অভিভাবকের সঙ্গে আলোচনা সাপেক্ষে। কিন্তু তারা ভবিষ্যতে যে বিষয়েই লেখাপড়া করুক না কেন, প্রোগ্রামিংয়ের এই জ্ঞান ও চর্চা তাদের নিঃসন্দেহে অন্যরকম মানুষ হিসেবে গড়ে তুলবে। তবে একথা বলে নেওয়া দরকার যে, ডেটা স্ট্রাকচার ও অ্যালগরিদম বিষয়টির ব্যাপ্তি অনেক বেশি। এই বইতে কেবল খুব প্রচলিত ও সহজ কিছু ডেটা স্ট্রাকচার ও অ্যালগরিদমের সঙ্গে শিক্ষার্থীদের পরিচয় করিয়ে দেওয়ার চেষ্টা করেছি। ভবিষ্যতে তারা কম্পিউটার বিজ্ঞান বিষয়ে আরো লেখাপড়া করলে, আরো অনেক ডেটা স্ট্রাকচার ও অ্যালগরিদম সম্পর্কে জানতে পারবে।

এই বইটি কম্পিউটার প্রোগ্রামিং সিরিজের তৃতীয় ও শেষ খণ্ড। ভবিষ্যতে এই সিরিজে আমার আর কোনো বই লেখার পরিকল্পনা নেই, তবে বর্তমান বইগুলোকে আরো উন্নততর করার প্রয়াস অব্যাহত রাখব। আর বাংলাদেশে বিভিন্ন বিশ্ববিদ্যালয়ে যেসব শিক্ষার্থী কম্পিউটার বিজ্ঞান পড়বে, তারা যদি স্কুল-কলেজে পড়ার সময়ই কম্পিউটার প্রোগ্রামিং বইয়ের তিনটি খণ্ড পড়ে ফেলে, তাহলে বাংলাদেশের প্রযুক্তির জগতে যে অভাবনীয় পরিবর্তন ঘটবে, তা ভাবতেই আমি শিহরিত হচ্ছি।

বইটি লেখার ব্যাপারে অনেকেই প্রত্যক্ষ ও পরোক্ষভাবে সহযোগিতা করেছেন। দ্বিমিক প্রকাশনীর তাহমিদ রাফি বইটি রিভিউ করার পাশাপাশি সম্পাদনাও করেছেন। এ ছাড়া শাহজালাল বিজ্ঞান ও প্রযুক্তি বিশ্ববিদ্যালয়ের শিক্ষক এবং বর্তমানে যুক্তরাষ্ট্রে পিএইচডি অধ্যয়নরত শহীদুল ইসলাম (সুমন) ও রুহুল আমীন (সজীব) অনেক গুরুত্বপূর্ণ মতামত দিয়েছেন। এ ছাড়া ন্যাশনাল ইউনিভার্সিটি অব সিঙ্গাপুরের কোয়ান্টাম কম্পিউটার বিজ্ঞানী তানভীরুল ইসলাম এবং বাংলাদেশের তরুণ সফটওয়্যার নির্মাতা আবু আশরাফ মাসনুনের কাছেও আমি কৃতজ্ঞ। বরাবরের মতো আমার স্ত্রী সিরাজুম মুনিরা (পারমিতা) বেশিরভাগ সময় আমাদের ছেলের দেখাশোনা করার দায়িত্ব নিজের ঘাড়ে নেওয়ায় বইটি লেখার জন্য আমার পক্ষে সময় বের করা সম্ভব হয়েছে। তাই আমার ও এই বইয়ের হবু পাঠকদের পক্ষ থেকে তাকেও ধন্যবাদ জানাই।

বাংলাদেশের তরুণরা এখনো শিক্ষায়, জ্ঞানে ও দক্ষতায় উন্নত বিশ্বের চেয়ে অনেক পিছিয়ে আছে। কিন্তু একদিন তারা বিশ্বমানের হবে। সেই অনাগত দিনের অপেক্ষায় রইলাম।

তামিম শাহরিয়ার সুবিন,
গ্র্যাব ইঞ্জিনিয়ারিং হেডকোয়ার্টার, সিঙ্গাপুর।

লেখক পরিচিতি



তামিম শাহ্‌রিয়ার (ডাকনাম সুবিন)-এর জন্ম 1982 সালের 7 নভেম্বর ময়মনসিংহে। গ্রামের বাড়ি কুমিল্লা জেলার চান্দিনা উপজেলার হারং গ্রামে। তাঁর বাবা মো. মোজাম্মেল হক ছিলেন সরকারি কর্মকর্তা এবং মা ফেরদৌসি বেগম গৃহিণী। স্ত্রী সিরাজুম মুনিরা ও পুত্র আরাভ শাহরিয়ারকে নিয়ে বর্তমানে সিঙ্গাপুরে বসবাস করছেন। তিনি বাংলা ভাষায় রচিত কয়েকটি জনপ্রিয় প্রোগ্রামিং বইয়ের লেখক।

লেখাপড়া করেছেন হোমনা সরকারি প্রাথমিক বিদ্যালয়, এ কে উচ্চ বিদ্যালয়, নটর ডেম কলেজ এবং শাহজালাল বিজ্ঞান ও প্রযুক্তি বিশ্ববিদ্যালয়ে। 2006 সালে শাহজালাল বিজ্ঞান ও প্রযুক্তি বিশ্ববিদ্যালয়ে কম্পিউটার সায়েন্স ও ইঞ্জিনিয়ারিং বিভাগ থেকে পাস করেছেন। বিশ্ববিদ্যালয়ে থাকাকালীন বিভিন্ন প্রোগ্রামিং প্রতিযোগিতায় অংশগ্রহণ করেছেন। পরবর্তী সময়ে (2007 ও 2008 সালে) তিনি এসিএম আইসিপিসি ঢাকা রিজিওনাল-এর বিচারক ছিলেন।

একটি বেসরকারি বিশ্ববিদ্যালয়ে শিক্ষকতা দিয়ে কর্মজীবন শুরু করলেও পরে একটি দেশি সফটওয়্যার নির্মাতা প্রতিষ্ঠানে কাজ করেন। তারপর যুক্তরাষ্ট্রভিত্তিক আরেকটি সফটওয়্যার তৈরির প্রতিষ্ঠানে কাজ করার পর নিজেই প্রতিষ্ঠা করেন মুক্ত সফটওয়্যার লিমিটেড নামক সফটওয়্যার তৈরির একটি প্রতিষ্ঠান। অনলাইন কোর্সের মাধ্যমে প্রোগ্রামিং ও সফটওয়্যার তৈরির নানা বিষয় শিক্ষাদানের জন্য তৈরি করেছেন দ্বিমিক কম্পিউটিং। এ ছাড়া তিনি বাংলাদেশ গণিত অলিম্পিয়াডে একজন একাডেমিক কাউন্সিলর। বর্তমানে ইঞ্জিনিয়ারিং ম্যানেজার হিসেবে কাজ করছেন সিঙ্গাপুরের গ্রাব (Grab) নামক প্রতিষ্ঠানে।

অধ্যায় ১ : ডেটা স্ট্রাকচার (Data Structure) ও অ্যালগরিদম (Algorithm)

কম্পিউটারে প্রোগ্রামিং করে আমাদেরকে বিভিন্ন ধরনের সমস্যার সমাধান করতে হয়। সেই সমস্যাগুলোর জটিলতা বিভিন্ন পর্যায়ের হতে পারে। একটি সমস্যা যোগ-বিয়োগ করার মতো সহজ যেমন হতে পারে, তেমনি একটি শহরের রাস্তাঘাট ও সেখানে যানবাহন চলাচলের তথ্য বিশ্লেষণ করে কোথায় কোথায় বিকল্প সড়ক তৈরি করতে হবে, সেটি বের করার মতো জটিল হতে পারে। সমস্যা সহজ হোক কিংবা জটিল, আমরা কম্পিউটার প্রোগ্রাম লিখে সেই সমস্যা সমাধান করার চেষ্টা করি। একটি সমস্যা সমাধান করার জন্য যখন আমরা প্রোগ্রাম তৈরি করি, সেই প্রোগ্রামটির কেবল সঠিক ফলাফল দিলেই হবে না, সেই সঙ্গে গ্রহণযোগ্য সময়ের মধ্যে ফলাফল দিতে হবে। বিভিন্ন সমস্যা সমাধানের জন্য সঠিক ও কার্যকর প্রোগ্রাম তৈরি করে সেই সমস্যা যেন সমাধান করা যায়, সেজন্য ডেটা স্ট্রাকচার ও অ্যালগরিদম জানতে ও শিখতে হয়।

ডেটা স্ট্রাকচার (Data Structure)

ডেটা স্ট্রাকচার হচ্ছে কম্পিউটারের মেমোরিতে বিভিন্নভাবে ডেটা সাজানো ও সংরক্ষণ করার পদ্ধতি। আমরা ইতিমধ্যে দেখেছি যে, কিছু কিছু প্রোগ্রামে ডেটা নিয়ে কাজ করতে গেলে কেবল ভ্যারিয়েবল ব্যবহার করলেই চলে, আবার কিছু কিছু প্রোগ্রামে আমাদেরকে স্ট্রাকচার ব্যবহার করতে হয়। আবার একই ধরনের অনেক ডেটা নিয়ে কাজ করতে গেলে অ্যারে ব্যবহার করতে হয়। কিন্তু, যখন আমরা আরো বড়ো পরিসরে কাজ করব, কিংবা, বিশেষ ধরনের কোনো কাজ করব, তখন আমাদের আলাদাভাবে ডেটা সাজাতে হবে ও সংরক্ষণ করতে হবে, ডেটাগুলোর মধ্যে সম্পর্ক তৈরি করতে হবে, ডেটার ওপর বিভিন্ন অপারেশনের জন্য বিশেষ ব্যবস্থা গ্রহণ করতে হবে। এজন্য আমাদেরকে বিভিন্ন ডেটা স্ট্রাকচার ব্যবহার করতে হবে।

ইতিমধ্যেই কিন্তু আমরা একটি ডেটা স্ট্রাকচারের ব্যবহার সবাই করেছি, সেটি হচ্ছে অ্যারে। কম্পিউটার বিজ্ঞানে অনেক রকমের ডেটা স্ট্রাকচার রয়েছে। এই বইতে আমরা মোটামুটি সহজ ধরনের কিছু ডেটা স্ট্রাকচার শিখব। এসব ডেটা স্ট্রাকচার সব প্রোগ্রামারেরই জানা উচিত। প্রতিটি ডেটা স্ট্রাকচার সম্পর্কে পড়ার সময় আমরা শেখার চেষ্টা করব যে, ডেটা স্ট্রাকচারটি কীভাবে তৈরি করতে হয়, কীভাবে একটি প্রোগ্রামে ব্যবহার করতে হয় এবং সেই ডেটা স্ট্রাকচারের বিভিন্ন

অপারেশনের টাইম ও স্পেস কমপ্লেক্সিটি কেমন। টাইম ও স্পেস কমপ্লেক্সিটি বিষয়ে পরের অধ্যায়ে বিস্তারিত আলোচনা করব।

অ্যালগরিদম (Algorithm)

অ্যালগরিদম হচ্ছে সমস্যা সমাধানের জন্য কিছু নির্দিষ্ট ধাপ। এই ধাপগুলো আবার গাণিতিকভাবে প্রমাণিত। অর্থাৎ, একটি সমস্যা সমাধানের জন্য যেই অ্যালগরিদম দেওয়া হয়, সেটি গাণিতিকভাবে প্রমাণ করতে হয়। তবে, এই বইতে আমরা গাণিতিক প্রমাণ শিখব না, বরং, বিভিন্ন ধরনের অ্যালগরিদমের সঙ্গে পরিচিত হব। কম্পিউটার বিজ্ঞানে অসংখ্য অ্যালগরিদম রয়েছে, তাদের মধ্যে থেকে কেবল প্রাথমিক কিছু অ্যালগরিদমের সঙ্গেই আমরা পরিচিত হব।

একটি অ্যালগরিদম ব্যবহার করে একটি নির্দিষ্ট সমস্যার সমাধান করা যায়। একটি নির্দিষ্ট সমস্যা একাধিক অ্যালগরিদম ব্যবহার করেও সমাধান করা যায়। আবার একটি অ্যালগরিদম দিয়ে একটি নির্দিষ্ট ধরনের বিভিন্নরকম সমস্যার সমাধান করা যায়। যেমন, ধরা যাক, সর্টিং সমস্যা—যেখানে অনেক ডেটার সমাহারকে একটি নির্দিষ্ট ক্রমে সাজাতে হয়। ডেটা যদি সংখ্যা জাতীয় হয়, তাহলে ছোটো থেকে বড়ো বা বড়ো থেকে ছোটো ক্রমে। আবার এমনও হতে পারে যে, পরীক্ষায় প্রাপ্ত নম্বরের ভিত্তিতে শিক্ষার্থীদের নাম ও প্রাপ্ত নম্বর ক্রমান্বয়ে সাজাতে হবে। অর্থাৎ, যে সবচেয়ে বেশি নম্বর পেয়েছে, তার নাম ও পরীক্ষায় প্রাপ্ত নম্বর থাকবে প্রথমে, তারপরে থাকবে, যে দ্বিতীয় সর্বোচ্চ নম্বর পেয়েছে, তার নাম ও পরীক্ষায় প্রাপ্ত নম্বর, এভাবে সাজাতে হবে। তো, এই সর্টিং সমস্যার সমাধান করার জন্যই ২০-৩০ রকমের অ্যালগরিদম আছে। তাদের মধ্যে কয়েকটি জনপ্রিয় অ্যালগরিদম হচ্ছে—ইনসার্শন সর্ট, কুইক সর্ট, মার্জ সর্ট ইত্যাদি। প্রতিটি অ্যালগরিদমেরই আবার কিছু সুবিধা ও অসুবিধা রয়েছে। তাই আমাদের সেগুলো সম্পর্কে ভালোভাবে জানতে হবে, যেন প্রোগ্রাম লেখার সময় আমরা আমাদের সমস্যার ধরন অনুযায়ী সবচেয়ে উপযুক্ত অ্যালগরিদম ব্যবহার করতে পারি।

ডেটা স্ট্রাকচার ও অ্যালগরিদম কেন শিখব?

ডেটা স্ট্রাকচার ও অ্যালগরিদম অপেক্ষাকৃত জটিল একটি বিষয়। তাই ধৈর্য ধরে এবং পূর্ণ মনোযোগ দিয়ে বইটি পড়তে হবে। ওপরের অংশটুকু পড়ার পরেও যদি কারো মনে হয়, কেন আমি কষ্ট করে ডেটা স্ট্রাকচার ও অ্যালগরিদম শিখব, তাদের জন্য আমি কিছু তথ্য দিচ্ছি—

- প্রোগ্রামিং প্রতিযোগিতায় ভালো করতে হলে ডেটা স্ট্রাকচার ও অ্যালগরিদম জানা থাকা লাগবেই। অনেক প্রোগ্রামিং সমস্যাই বিভিন্ন অ্যালগরিদম ও ডেটা স্ট্রাকচার ব্যবহার করে অনেক সহজে ও কম সময়ে সমাধান করা যায়।
- যারা কম্পিউটার বিজ্ঞান বিষয়ে পড়ছে কিংবা পড়বে, তাদের অ্যাকাডেমিক সিলেবাসে এই বিষয়ের ওপর একাধিক কোর্স আছে। সেসব কোর্সে ভালো করতে হলে ডেটা স্ট্রাকচার ও অ্যালগরিদম সম্পর্কে কিছু প্রাথমিক জ্ঞান থাকা প্রয়োজন।
- যারা অনেক ভালো মানের সফটওয়্যার প্রকৌশলী হতে চায়, তাদের ডেটা স্ট্রাকচার ও অ্যালগরিদম শিখতে হবে। কারণ, অনেক রকম সফটওয়্যার তৈরির কাজে এগুলো ব্যবহার করতে হবে।
- ডেটা স্ট্রাকচার ও অ্যালগরিদম যেহেতু তুলনামূলকভাবে একটু জটিল বিষয়, তাই এগুলো বুঝতে পারলে এবং এগুলো নিয়ে চিন্তাভাবনা করতে পারলে সমস্যা বিশ্লেষণ করার দক্ষতা বাড়ে। আর এই দক্ষতা একজন ভালো প্রোগ্রামার হতে গেলে অবশ্যই থাকতে হবে।
- দেশি-বিদেশি বিভিন্ন সফটওয়্যার নির্মাতা প্রতিষ্ঠানে সফটওয়্যার ইঞ্জিনিয়ার পদে নিয়োগের জন্য ইন্টারভিউতে ডেটা স্ট্রাকচার ও অ্যালগরিদমের জ্ঞান যাচাই করা হয়। তাই, দেশের শীর্ষ প্রতিষ্ঠানগুলোতে কাজ করতে হলে এবং আন্তর্জাতিক পর্যায়ে কাজ করতে হলে এই বিষয়ে দক্ষ হতে হবে।
- কিছু উচ্চতর বিষয়, যেমন মেশিন লার্নিং, কৃত্রিম বুদ্ধিমত্তা (আর্টিফিশিয়াল ইনটেলিজেন্স—Artificial Intelligence) ইত্যাদি নিয়ে কাজ করতে হলে ডেটা স্ট্রাকচার ও অ্যালগরিদম অবশ্যই জানতে হবে।
- এমনকি যারা স্কুল-কলেজে পড়ছে, তারা যদি ডেটা স্ট্রাকচার ও অ্যালগরিদম শেখে, তাহলে তারা ভবিষ্যতে যে বিষয়েই লেখাপড়া করুক না কেন, তাদের যৌক্তিকভাবে বিভিন্ন সমস্যা নিয়ে চিন্তা করা ও সমস্যা বিশ্লেষণ করার এই দক্ষতা (ইংরেজিতে বলে অ্যানালিটিকাল স্কিলস্ - Analytical Skills) অনেক কাজে আসবে।

অ্যালগরিদম লেখার বিভিন্ন পদ্ধতি আছে। একেবারে বর্ণনামূলকভাবে লেখা যেতে পারে, আবার প্রতিটি ধাপ আলাদাভাবে উল্লেখ করে লেখা যেতে পারে, আবার কখনো-বা সুডোকোড (Pseudocode) আকারে লেখা যেতে পারে। সুডোকোড হচ্ছে অনেকটা প্রোগ্রামের কোডের মতোই, তবে এই কোডের জন্য কোনো কম্পাইলার বা ইন্টারপ্রেটার নেই। মানুষের বোঝার জন্যই এই কোড। সুডোকোড দেখে প্রোগ্রামাররা তাদের পছন্দমতো প্রোগ্রামিং ভাষায় কোড লিখতে পারবে। অ্যালগরিদম মুখস্থ না করে ঠিকঠাকভাবে বুঝলে তা নিজের পছন্দসই ভাষায় লিখতে তেমন বেগ পেতে হবে না। অ্যালগরিদম কীভাবে কাজ করে তা ঠিকমতো বুঝে নিয়ে সেটি ইমপ্লিমেন্ট (implement) করতে পারতে হবে। আমরা এই বইতে সি (C) প্রোগ্রামিং ভাষা ব্যবহার করে ডেটা স্ট্রাকচার তৈরি করব এবং অ্যালগরিদম ইমপ্লিমেন্ট করব। একাডেমিক

প্রয়োজনে (মানে কোর্সের পরীক্ষায়) অ্যালগরিদম লিখতে হলে, একাডেমিক সিলেবাসে প্রস্তাবিত বই অনুসরণ করে অ্যালগরিদম লিখলেই সবচেয়ে ভালো হয়।

বইটি কাদের জন্য এবং কীভাবে পড়তে হবে?

যারা কম্পিউটার প্রোগ্রামিং ১ম খণ্ড ও ২য় খণ্ড বই দুটি পড়েছে, তাদের জন্যই এই বইটি লেখা। এ ছাড়া যে-কেউ, যার সি প্রোগ্রামিং ভাষা মোটামুটি জানা আছে, তার এই বইটি পড়ে বুঝতে অসুবিধা হবে না। বইটি পড়ার সময় প্রতিটি অ্যালগরিদম পুরোপুরি বোঝার চেষ্টা করতে হবে। একবার পড়ে না বুঝলে দু-তিনবার করে পড়তে হবে। তারপরে প্রতিটি কোড নিজে দেখে দেখে ইমপ্লিমেন্ট করতে হবে। অনুশীলনীগুলো নিজে করার চেষ্টা করতে হবে। কোনো সমস্যা হলে বন্ধুদের সঙ্গে সমাধান নিয়ে আলোচনা করতে হবে। এ ছাড়া বইটি পড়ার পরে, বিভিন্ন ওয়েবসাইটে সংশ্লিষ্ট সমস্যাগুলো সমাধান করার চেষ্টা করলে শেখাটা আরো ভালো হবে। এ ছাড়া কোনো প্রশ্ন থাকলে *প্রোগ্রামাবাদ* ওয়েবসাইটে প্রথমে সার্চ করতে হবে এবং সেই রকম প্রশ্ন খুঁজে না পেলে তখন প্রশ্ন করতে হবে। ওয়েবসাইটের লিংক হচ্ছে— http://programabad.com

সবশেষে, পুরো বইটি একবার পড়া শেষ হলে আরেকবার ভালোভাবে পড়তে হবে। দ্বিতীয়বার পড়ার সময়ও সব কোড নিজে নিজে করতে হবে।

অধ্যায় ২ : টাইম ও স্পেস কমপ্লেক্সিটি (Time & Space Complexity)

আগের অধ্যায়ে আমরা বলেছি, একটি সমস্যার সমাধানের জন্য একাধিক অ্যালগরিদম থাকতে পারে। তো আমরা কখন কোন অ্যালগরিদম ব্যবহার করব? আমাদের নির্দিষ্ট সমস্যাটি সমাধানের জন্য কোন অ্যালগরিদম সবচেয়ে ভালো হবে, কেন ভালো হবে? এই প্রশ্নের উত্তর খুঁজে পেতে হলে, জানতে হবে টাইম ও স্পেস কমপ্লেক্সিটি (Time & Space Complexity)।

টাইম কমপ্লেক্সিটি (Time Complexity)

আমরা কেন নিজেরা কাজ না করে কম্পিউটারকে দিয়ে কাজ করাই? কারণ, কম্পিউটার খুব দ্রুত এবং নির্ভুলভাবে কাজ করতে পারে। যেমন, এক লক্ষ সংখ্যা হাতে হাতে যোগ করতে হলে আমাদের কয়েক দিন সময় লেগে যাবে। কিন্তু, কম্পিউটারকে সেই কাজ করতে দিলে এক সেকেন্ডের কম সময়ে সে কাজটি করে ফেলবে। কাজটি করার জন্য যদি কম্পিউটারেরও কয়েক দিন সময় লেগে যেত, তাহলে তো আমরা কম্পিউটার ব্যবহার করতাম না। তাই, একটি কাজ করতে একটি প্রোগ্রামের কতটুকু সময় লাগল, সেটি আমাদের কাছে খুবই গুরুত্বপূর্ণ। এখানেই চলে আসে টাইম কমপ্লেক্সিটির বিষয়টি। আমরা তো প্রোগ্রামে সময়সংক্রান্ত লাইব্রেরি ফাংশন ব্যবহার করে বের করতে পারি যে, প্রোগ্রাম রান করতে কতটুকু সময় লাগল। কিন্তু, আমরা যদি টাইম কমপ্লেক্সিটির হিসাবকিতাব বুঝতে পারি, তখন আমাদের প্রোগ্রাম রান করে সেটি বের করতে হবে না, একটি অ্যালগরিদম দেখলে বা একটি প্রোগ্রাম দেখলেই আমরা বলে দিতে পারব যে, এই প্রোগ্রাম যদি রান করা হয়, তাহলে কেমন সময় লাগতে পারে।

স্পেস কমপ্লেক্সিটি (Space Complexity)

কম্পিউটারকে দিয়ে কাজ করানোর আরেকটি কারণ হচ্ছে, যন্ত্রটি অনেক কিছু ঠিকঠাকভাবে মনে রাখতে পারে। এর স্থায়ী ও অস্থায়ী মেমোরি রয়েছে। আমরা মানুষেরা অনেক সময় অনেক কিছু ভুলে যাই, আবার কারো স্মরণশক্তি বেশি, কারো কম। কিন্তু, কম্পিউটারের বেলাতে সেই কম্পিউটারে যতটুকু মেমোরি দেওয়া হয়, তার তথ্য ধারণক্ষমতা ততটুকুই থাকে। তাই কোনো কিছু মনে রাখা, অর্থাৎ, তথ্য ধারণ করার কাজটি আমরা কম্পিউটারের ওপর ছেড়ে দিতে চাই।

কম্পিউটারের মেমোরি অনেক বেশি হলেও একটি নির্দিষ্ট সীমা আছে। তাই আমাদেরকে প্রোগ্রাম লেখার সময় খেয়াল রাখতে হয় যে, প্রোগ্রামটি কতটুকু মেমোরি ব্যবহার করছে। প্রোগ্রাম চলাকালীন সেটি কতটুকু মেমোরি ব্যবহার করছে তা পর্যবেক্ষণ করা যায়। কিন্তু, আমরা যদি স্পেস কমপ্লেক্সিটির হিসাবনিকাশ বুঝি, তাহলে অ্যালগরিদম বা প্রোগ্রাম দেখে এবং সেটি বিশ্লেষণ করে সহজেই বলে দেওয়া সম্ভব যে, প্রোগ্রামটি কতটুকু মেমোরি ব্যবহার করবে।

একজন প্রোগ্রামারের চেষ্টা থাকা উচিত, তাঁর প্রোগ্রামের টাইম কমপ্লেক্সিটি ও স্পেস কমপ্লেক্সিটি—দুটোই যেন কম থাকে। তবে কখনো কখনো সময় বাঁচানোর জন্য অতিরিক্ত মেমোরি ব্যবহার করা হয়, আবার কখনো কখনো মেমোরি বাঁচানোর জন্য এমনভাবে প্রোগ্রাম তৈরি করা হয়, যেখানে অতিরিক্ত সময় খরচ করতে হয়। তো এই টাইম ও স্পেস কমপ্লেক্সিটি পরিমাপের জন্য বিভিন্ন গাণিতিক ফাংশন আছে, থেটা (θ), ওমেগা (Ω) এবং বিগ-ও (O)। কম্পিউটার সায়েন্সের শিক্ষার্থীরা ডিসক্রিট ম্যাথমেটিকস (Discrete Mathematics) কোর্সে এই বিষয়ে বিস্তর জ্ঞানলাভ করতে পারে। তবে আমরা জটিল গাণিতিক হিসাবনিকাশের মধ্যে না গিয়ে একটু সহজ-সরল উপায়ে কমপ্লেক্সিটি শিখব, তাই আমাদের আপাতত বিগ-ও (O) জানলেই চলবে। আমরা এখন কিছু উদাহরণ দেখব।

কিছু উদাহরণ

ধরা যাক, আমাকে দুটি সংখ্যা যোগ করার একটি প্রোগ্রাম লিখতে হবে। প্রোগ্রামটি এমন :

```
#include <stdio.h>

int main()
{
    int n1, n2, result;

    n1 = 10;
    n2 = 20;
    result = n1 + n2;

    return 0;
}
```

এখন প্রশ্ন হচ্ছে, ওপরের প্রোগ্রামটির কমপ্লেক্সিটি কত? বলে নেওয়া ভালো যে, প্রোগ্রামটি রান করলে তোমরা কোনো আউটপুট দেখবে না, কারণ আমি কোনো কিছু প্রিন্ট করিনি।

> ◉ আমরা যখন শুধু কমপ্লেক্সিটি কথাটি বলব, তখন সাধারণত টাইম কমপ্লেক্সিটিকে বোঝাব। আর স্পেস কমপ্লেক্সিটির বেলাতে পুরোটি কথাটিই লিখব। চাকরির ইন্টারভিউয়ে কমপ্লেক্সিটি নিয়ে প্রশ্ন করা হলে, পালটা প্রশ্ন করে জেনে নিতে হবে যে, টাইম কমপ্লেক্সিটির কথা জানতে চাওয়া হচ্ছে, নাকি স্পেস কমপ্লেক্সিটি।

প্রোগ্রামটিতে চারটি কাজ করা হচ্ছে—

1. n1 ভ্যারিয়েবলের ভেতরে 10 রাখা।
2. n2 ভ্যারিয়েবলের ভেতরে 20 রাখা।
3. n1 ও n2 ভ্যারিয়েবল দুটি যোগ করা।
4. যোগফলটি `result` নামক ভ্যারিয়েবলে রাখা।

n1 ও n2 ভ্যারিয়েবলের যেকোনো মানের ক্ষেত্রেই কিন্তু কম্পিউটার ওপরের চারটি অপারেশন করবে (একটি যোগ অপারেশন আর তিনটি অ্যাসাইনমেন্ট অপারেশন)। এরকম ক্ষেত্রে আমরা প্রোগ্রামটির কমপ্লেক্সিটি বলতে পারি $O(1)$ (পড়তে হবে, অর্ডার অব ওয়ান)। এ ক্ষেত্রে চারটির বদলে একটি কিংবা দশটি অপারেশন হলেও আমরা কমপ্লেক্সিটি $O(1)$-ই লিখতাম। তাহলে আমরা বলতে পারি, যেকোনো ইনপুটের ক্ষেত্রেই যদি আমাদের প্রোগ্রামে অপারেশনের সংখ্যা ধ্রুবক বা কনস্ট্যান্ট (constant) হয়, তাহলে সেই প্রোগ্রামের কমপ্লেক্সিটিকে আমরা বলব $O(1)$।

এখন আমরা আরেকটি প্রোগ্রাম লিখব, যার কাজ হচ্ছে 1 থেকে n পর্যন্ত প্রতিটি পূর্ণসংখ্যা যোগ করা। এই প্রোগ্রামটি দুইভাবে লেখা যায়। ধারার যোগফলের সূত্র ব্যবহার করে, আর লুপের সাহায্যে। আমরা যারা গাণিতিক ধারার সঙ্গে পরিচিত (নবম-দশম শ্রেণির গণিত বইতে আছে), তারা জানি, 1 থেকে n পর্যন্ত প্রতিটি পূর্ণসংখ্যার যোগফল হচ্ছে $n \times \frac{n+1}{2}$। তাহলে আমরা প্রোগ্রামটি লিখতে পারি এভাবে—

```
#include <stdio.h>

int main()
{
    int n, result;

    scanf("%d", &n);

    result = n * (n + 1) / 2;

    printf("result = %d\n", result);
```

```
    return 0;
}
```

ওপরের প্রোগ্রামে আমরা প্রথমে n-এর মান ইনপুট নিচ্ছি, তাই কমপ্লেক্সিটি হিসাব করার সময় এই কাজটি আমরা বিবেচনা করব না। একই কারণে `printf()` ফাংশনটির কাজও বিবেচনায় আনা হবে না (এটি তোমাদের সুবিধার জন্য দিয়েছি, যেন তোমরা আউটপুট দেখতে পারো)। মূল কাজটি আমরা এক লাইনে করেছি (9 নম্বর লাইন), যেখানে আমরা একটি যোগ, একটি গুণ, একটি ভাগ ও একটি অ্যাসাইনমেন্ট অপারেশন করছি। এখন n-এর মান যা-ই হোক না কেন, আমরা এই চারটি অপারেশনই করব। তাই এ ক্ষেত্রে প্রোগ্রামের কমপ্লেক্সিটি হবে $O(1)$।

আমরা সূত্র ব্যবহার না করে লুপ চালিয়েও প্রোগ্রামটি লিখতে পারি। সে ক্ষেত্রে প্রোগ্রামটি হতে পারে এরকম—

```
int i, n, result;

scanf("%d", &n);

result = 0;

for (i = 1; i <= n; i++) {
    result = result + i;
}
```

এখানে, ইনপুট নেওয়ার পরে, প্রথমে একটি অ্যাসাইনমেন্ট অপারেশন হচ্ছে। তারপরে, n-এর মান যত, ততটি যোগ ও অ্যাসাইনমেন্ট অপারেশন হবে। সেই সঙ্গে nসংখ্যক তুলনা বা কম্পারিজন অপারেশনও চলবে (`i <= n` লাইনে)। এখন n-এর মান যদি 10 হয়, তাহলে লুপ চলবে 10 বার এবং 10 বার `result = result + i` স্টেটমেন্টটি চলবে, তাহলে 10টি যোগ ও 10টি অ্যাসাইনমেন্ট অপারেশন হবে। সঙ্গে 10টি কম্পারিজন অপারেশনও হবে কারণ লুপ ঘোরার সময় প্রতিবার `i <= n` শর্তটি পরীক্ষা করা হবে। তবে যেহেতু এটি কেবল লুপ ঘোরানোর কাজে ব্যবহার করা হচ্ছে, তাই আমরা এটি হিসাব করব না। কারণ এটি হিসাবে ধরলেও কমপ্লেক্সিটির কোনো পরিবর্তন হবে না।

n-এর মান যদি 1 হয়, তাহলে একটি যোগ ও একটি অ্যাসাইনমেন্ট অপারেশন হবে। আর n-এর মান 100 হলে 100টি যোগ ও 100টি অ্যাসাইমেন্ট অপারেশন হবে। তাহলে আমরা বলতে পারি, এই প্রোগ্রামে অপারেশনের সংখ্যা, n-এর মানের সমানুপাতিক। এ ক্ষেত্রে প্রোগ্রামটির কমপ্লেক্সিটি হবে $O(n)$ (পড়তে হবে, অর্ডার অব এন)।

তাহলে আমরা দেখলাম, একই প্রোগ্রাম বিভিন্নভাবে লেখা যায় এবং তাদের কমপ্লেক্সিটিও বিভিন্ন রকম হতে পারে। এখন, এই যে দুই রকমের কমপ্লেক্সিটি আমরা দেখলাম, এর তাৎপর্য কী? ধরা যাক, একটি যোগ ও একটি অ্যাসাইমেন্ট অপারেশন করতে এক মাইক্রোসেকেন্ড সময় লাগে। তাহলে, যেই প্রোগ্রামের কমপ্লেক্সিটি O(1), সেখানে n-এর মান 1 লক্ষ হলেও সেই প্রোগ্রামটি চলতে এক মাইক্রোসেকেন্ড সময় লাগবে। আর যেই প্রোগ্রামের কমপ্লেক্সিটি O(n), সেখানে n-এর মান 1 লক্ষ হলে প্রোগ্রামটি চলতে 1 লক্ষ মাইক্রোসেকেন্ড (বা, 100 মিলিসেকেন্ড) সময় লাগবে।

এবারে আসি স্পেস কমপ্লেক্সিটির একটি উদাহরণে। ধরা যাক, আমাকে বলা হলো একটি প্রোগ্রাম লিখতে, যেখানে একটি পূর্ণসংখ্যা n ইনপুট দেওয়া হবে, আর n জোড় না বিজোড়, সেটি বের করতে হবে। সেই সঙ্গে এটিও বলে দেওয়া হলো যে n-এর মান হবে 0 থেকে 100-এর মধ্যে। আমি প্রোগ্রামটি এভাবে লিখতে পারি—

```
#include <stdio.h>

int main()
{
    int n;

    scanf("%d", &n);

    if (n % 2 == 0) {
        printf("%d is even number.\n", n);
    } else {
        printf("%d is odd number.\n", n);
    }

    return 0;
}
```

ওপরের প্রোগ্রামটির টাইম কমপ্লেক্সিটি কত? যারা এতক্ষণ মনোযোগ দিয়ে বইটি পড়েছে, তাদের বুঝতে কষ্ট হবে না যে, এই প্রোগ্রামের টাইম কমপ্লেক্সিটি হচ্ছে O(1)। আর এই প্রোগ্রামের স্পেস কমপ্লেক্সিটিও হচ্ছে O(1)। কারণ, n-এর মান যা-ই দেওয়া হোক না কেন, মাত্র একটি ভ্যারিয়েবল ব্যবহার করলেই হচ্ছে। n-এর মান 0 হলেও একটি ভ্যারিয়েবল, আর n-এর মান 100 হলেও একটি ভ্যারিয়েবল। তাই, আমরা বলতে পারি, স্পেস কমপ্লেক্সিটি হচ্ছে O(1)। স্পেস কমপ্লেক্সিটির সঙ্গে ভ্যারিয়েবলের সম্পর্ক কী? আমরা যখন কোনো ভ্যারিয়েবল ব্যবহার করি, তখন

সেটি কম্পিউটারের মেমোরি ব্যবহার করে। তাই, কতটুকু মেমোরি ব্যবহৃত হলো, সেটি নির্ভর করে প্রোগ্রামে কতগুলো ভ্যারিয়েবল ব্যবহার করা হলো তার ওপর।

এখন আমরা জোড়-বিজোড় বের করার প্রোগ্রামটি অন্যভাবে লিখি—

```
#include <stdio.h>

int main()
{
    int i, n, even[101];

    for (i = 0; i < 101; i++) {
        even[i] = 0;
    }

    for (i = 0; i < 101; i += 2) {
        even[i] = 1;
    }

    scanf("%d", &n);

    if (even[n]) {
        printf("%d is even number.\n", n);
    } else {
        printf("%d is odd number.\n", n);
    }

    return 0;
}
```

এখানে আমি even নামে একটি অ্যারে তৈরি করলাম, যেখানে মোট 101টি উপাদান আছে। প্রথমে একটি লুপ চালিয়ে প্রতিটি উপাদানের মান 0 করে দিলাম। তারপর একটি লুপের সাহায্যে i-এর মান 0 থেকে শুরু করে 101-এর আগ পর্যন্ত 2 করে বাড়িয়ে গেলাম (0, 2, 4, ..., 100) এবং even[i]-এর ভেতর 1 রাখলাম। লুপ শেষ হওয়ার পরে সব জোড় সংখ্যার ক্ষেত্রে even অ্যারের ওই অবস্থানের উপাদানগুলোর মান হবে 1, আর বাদবাকি উপাদানগুলোর মান হবে 0। চাইলে লুপ চালিয়ে even অ্যারেটি প্রিন্ট করেও দেখা যেতে পারে। এভাবে প্রোগ্রাম লেখার সুবিধা কী? একবার লুপ চালিয়ে কোনগুলো জোড় আর কোনগুলো বিজোড়, সেটি বের করার পরে আমাদের প্রতিবার আর ভাগশেষের সাহায্যে জোড়-বিজোড় বের করতে হবে না। আমরা যেকোনো সংখ্যা m-এর জন্য even[m]-এর মান পরীক্ষা করব। even[m]-এর মান যদি 1 হয়, তাহলে

m-হচ্ছে জোড় সংখ্যা, আর তা না হলে বিজোড়। কিন্তু এই প্রোগ্রামের স্পেস কমপ্লেক্সিটি কত? আমরা দেখতে পাচ্ছি, n-এর মান 100 হলে আমাদেরকে একটি অ্যারে তৈরি করতে হয়েছে, যেখানে 101টি উপাদান রয়েছে। আবার n-এর মান যদি 10 হতো, তাহলে অ্যারেতে 11টি উপাদান থাকত। n-এর মান যদি 1000 হতো, তাহলে অ্যারেতে 1001টি উপাদান থাকত। তাহলে আমরা দেখছি, আমাদের অ্যারের উপাদান সংখ্যা n-এর মানের সমানুপাতিক। n-এর মান বাড়লে অ্যারের উপাদান সংখ্যাও সমানুপাতিক হারে বাড়ে। তাই প্রোগ্রামটির স্পেস কমপ্লেক্সিটি হচ্ছে $O(n)$। প্রোগ্রামটির টাইম কমপ্লেক্সিটিও হচ্ছে $O(n)$, কারণ শুরুতেই আমাদের n-এর সর্বোচ্চ মান (এক্ষেত্রে 101) পর্যন্ত একটি লুপ চালাতে হয়েছে। ।

এখন নেস্টেড লুপ ব্যবহার করে আমরা একটি প্রোগ্রাম লিখব।

```
#include <stdio.h>

int main()
{
    int i, j, n, count;

    scanf("%d", &n);

    count = 0;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            count = count + 1;
        }
    }

    printf("n = %d, count = %d\n", n, count);

    return 0;
}
```

প্রোগ্রামটি একাধিকবার রান করে দেখতে হবে যে, n-এর বিভিন্ন মানের জন্য আউটপুট কী আসে। ইনপুট হিসেবে তুমি n-এর মান 1, 2, 3, 10, 100 ইত্যাদি দিয়ে দেখতে পারো। তুমি দেখবে যে, n-এর মান বাড়ার সঙ্গে সঙ্গে count-এর মানও বাড়ছে, তবে এবারে সমানুপাতিক হারে বাড়ছে না; বরং, n-এর তুলনায় count-এর মান অনেক দ্রুত বাড়ছে। আমরা n ও count-এর বিভিন্ন মান নিয়ে একটি তালিকা তৈরি করতে পারি :

n	count
1	1
2	4
3	9
10	100

ওপরের তালিকাটি একটু ভালোভাবে লক্ষ করলে দেখতে পাবে যে, n ও count-এর মধ্যে একটি সম্পর্ক রয়েছে। count হচ্ছে n-এর বর্গ, অর্থাৎ n^2 (বা $n \times n$)। তাহলে আমরা যদি n ইনপুট দেই, count = count + 1 স্টেটমেন্টটি n^2সংখ্যক বার চলে। তাহলে, এই প্রোগ্রামের কমপ্লেক্সিটি হচ্ছে $O(n^2)$।

অনুশীলনী :

নিচের প্রোগ্রামের কমপ্লেক্সিটি কত?

```
#include <stdio.h>

int main()
{
    int i, n, count;

    scanf("%d", &n);

    count = 0;

    for (i = 0; i < n; i++) {
        count = count + 1;
    }

    printf("n = %d, count = %d\n", n, count);

    return 0;
}
```

এখন, অ্যালগরিদমের বা প্রোগ্রামের কমপ্লেক্সিটি যেমন n-এর বর্গ হতে পারে, তেমনি n-এর ঘনও (অর্থাৎ n^3) হতে পারে। এ রকম কখন হবে? একটু চিন্তাভাবনা করলেই বুঝতে পারবে, যদি তিনটি লুপ নেস্টেড অবস্থায় থাকে, অর্থাৎ একটির ভেতরে আরেকটি থাকে, তখন কমপ্লেক্সিটি হবে $O(n^3)$। যেমন—

```
#include <stdio.h>

int main()
{
    int i, j, k, n, count;

    scanf("%d", &n);

    count = 0;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            for (k = 0; k < n; k++) {
                count = count + 1;
            }
        }
    }

    printf("n = %d, count = %d\n", n, count);

    return 0;
}
```

এখন তোমাদের কাজ হচ্ছে, প্রোগ্রামটি রান করে n-এর বিভিন্ন মান ইনপুট দিয়ে আউটপুট দেখা এবং n-এর মানের সঙ্গে count-এর মানের সম্পর্ক বের করা।

অনুশীলনী :

একজন প্রোগ্রামারকে দুটি অ্যালগরিদম দেওয়া হলো যাদের স্পেস কমপ্লেক্সিটি সমান। কিন্তু প্রথমটির টাইম কমপ্লেক্সিটি হচ্ছে $O(n^2)$ আর দ্বিতীয়টির টাইম কমপ্লেক্সিটি হচ্ছে $O(n^3)$। তাহলে তার কোন অ্যালগরিদমটি ব্যবহার করা উচিত এবং কেন?

একটি বিষয় আমি তোমাদের কাছে পরিষ্কার করতে চাই। কোনো প্রোগ্রামে একটি স্টেটমেন্ট থাকলেও কমপ্লেক্সিটি $O(1)$, পাঁচটি স্টেটমেন্ট থাকলেও কমপ্লেক্সিটি $O(1)$। $O(1)$ দিয়ে আমরা

আসলে একটি অপারেশন বোঝাই না, বরং বোঝাই ধ্রুবক (constant) অপারেশন। অর্থাৎ, n-এর মানের ওপর অপারেশনের সংখ্যা নির্ভর করে না। এখন, বাকি প্রোগ্রামের কমপ্লেক্সিটি যদি $O(n)$ হতো, তাহলে মোট কমপ্লেক্সিটি কত হতো? $O(n) + O(1)$। এখন, $O(1)$, $O(n)$-এর তুলনায় অনেক ছোটো বা অতি নগণ্য। তাই একে আমরা গণ্য করব না। তাই $O(n) + O(1) = O(n)$ লিখব। তেমনি, $O(n^4) + O(n^3) + O(n^2) + O(n) + O(1)$ (একে $O(n^4 + n^3 + n^2 + n + 1)$-ও লেখা যায়)-কে আমরা লিখব $O(n^4)$। কারণ, n-এর মান বাড়ার সঙ্গে সঙ্গে, বাকি রাশিগুলোর মান $O(n^4)$-এর তুলনায় নগণ্য হয়ে যায়।

অনুশীলনী :

নিচের প্রোগ্রামটির কমপ্লেক্সিটি কত?

```
#include <stdio.h>

int main()
{
    int i, j, n, count;

    scanf("%d", &n);

    count = 0;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            count = count + 1;
        }
    }

    for (i = 0; i < n; i++) {
        count = count + 1;
    }

    printf("count = %d\n", count);

    return 0;
}
```

অধ্যায় ৩ : সার্চিং অ্যালগরিদম (Searching Algorithm)

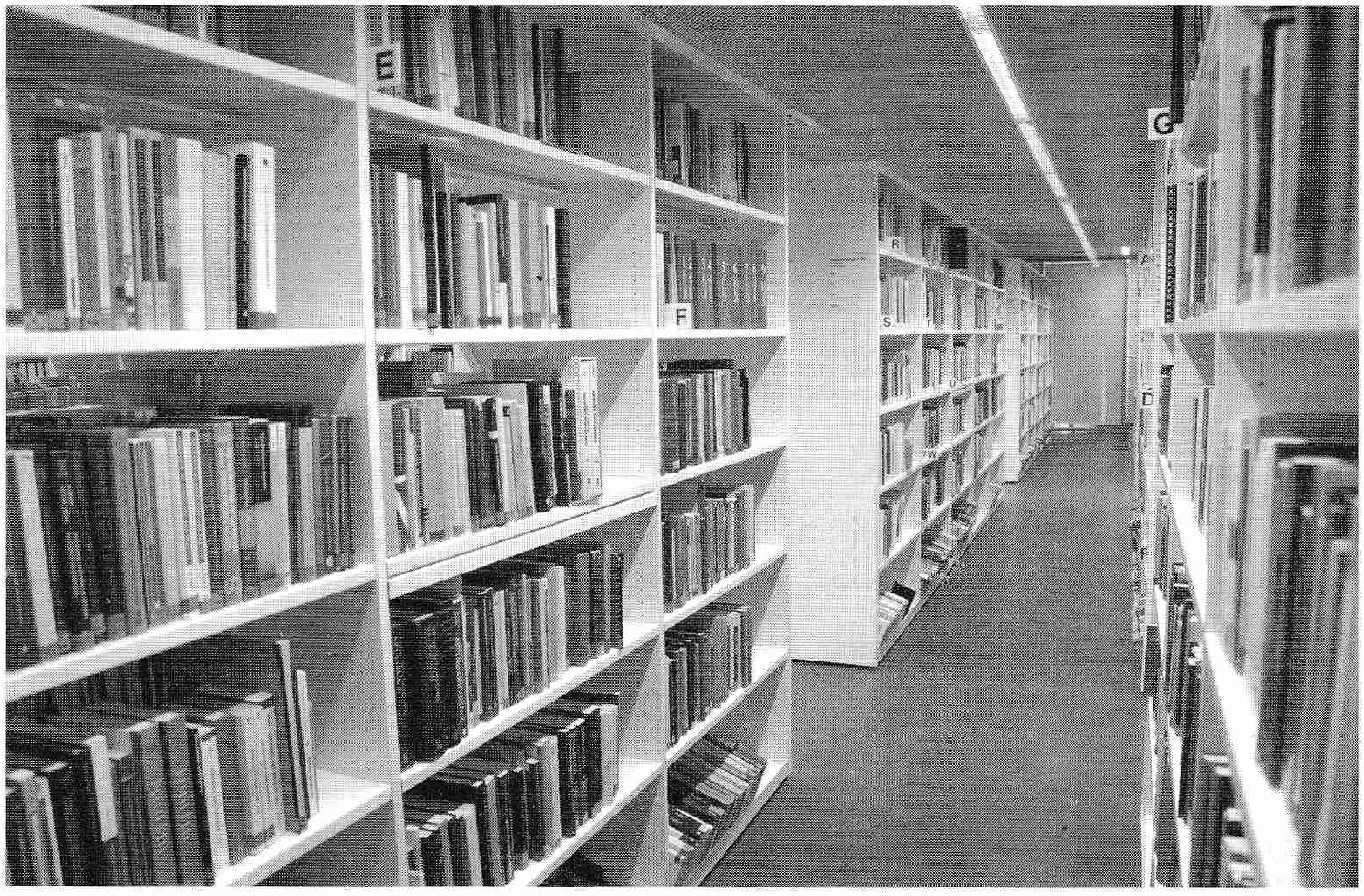
কম্পিউটারের বিভিন্ন প্রোগ্রাম বিভিন্ন রকমের কাজ করে। তাদের মধ্যে একটি কাজ অনেক প্রোগ্রামকেই করতে হয়, সেটি হচ্ছে অনেকগুলো জিনিসের মধ্যে থেকে নির্দিষ্ট কোনো জিনিস খুঁজে বের করা। এই খুঁজে বের করাকে ইংরেজিতে বলে search (সার্চ), তাই খোঁজাখুঁজিসংক্রান্ত অ্যালগরিদমকে বলা হয় সার্চিং অ্যালগরিদম (searching algorithm)। আমরা এই অধ্যায়ে দুটি খুব সহজ ও প্রচলিত সার্চিং অ্যালগরিদম সম্পর্কে জানব। তোমরা যারা কম্পিউটার প্রোগ্রামিং ১ম খণ্ড বইটি পড়েছ, তারা সবাই এই দুটি অ্যালগরিদমের সঙ্গে ইতিমধ্যে পরিচিত। তার পরও এগুলোর গুরুত্ব এতই বেশি যে, আমি এগুলো দিয়েই আলোচনা শুরু করতে চাই।

লিনিয়ার সার্চ (Linear search)

একটি বুকশেলফে যদি পাঁচটি বই থাকে, আর সেখান থেকে আমাকে একটি নির্দিষ্ট বই খুঁজে বের করতে বলা হয়, তাহলে কিন্তু আমি চট করেই সেটি খুঁজে পাব। এই কাজটি করার জন্য আপাতদৃষ্টিতে আমাকে তেমন চিন্তাভাবনাও করতে হবে না। কিন্তু বুকশেলফে যদি এক হাজার বই থাকে, আর সেখান থেকে আমাকে একটি নির্দিষ্ট বই খুঁজতে বলা হয়, তখন কাজটি আমার জন্য একটু কঠিন হয়ে যাবে।

এখন, আমি যদি এলোমেলোভাবে বিভিন্ন তাকে বইটি খুঁজতে থাকি, তাহলে আমি বইটি দ্রুত খুঁজে পেলেও পেতে পারি, আবার এমনও হতে পারে যে, সারা দিন খুঁজেও বইটি পেলাম না। তাই এলোমেলোভাবে খুঁজলে আমি যে বইটি খুঁজে পাবই, তার কোনো নিশ্চয়তা নেই। তাই আমি দ্রুত বই খুঁজে পাওয়ার আশায় এলোমেলো বা বিক্ষিপ্তভাবে বইটি না খুঁজে যদি প্রথম তাকের প্রথম বই থেকে একে একে খোঁজা শুরু করি, তাহলে একসময় বইটি পেয়ে যাব। ভাগ্য ভালো হলে খোঁজার শুরুতেই বইটি পেয়ে যাব (যদি বইটি প্রথম তাকের প্রথম বই হয়)। আর ভাগ্য একটু খারাপ হলে বইটি আমি পাব শেষ তাকের একেবারে শেষে। কিন্তু এভাবে বইটি খোঁজার সুবিধা হচ্ছে, বইটি আমি খুঁজে পাবই সেটা নিশ্চিত। আর যদি খুঁজে না পাই, তখন নিশ্চিতভাবে বলতে পারব যে, বইটি বুকশেলফে নেই। এভাবে একেবারে শুরু থেকে একটি একটি করে খোঁজার প্রক্রিয়াকে কম্পিউটার বিজ্ঞানের ভাষায় বলে লিনিয়ার সার্চ, বাংলায় আমরা বলতে পারি, বরাবর খোঁজা বা একাধারে খোঁজা।

ওপরের উদাহরণে যদি আমি বুকশেলফকে একটি অ্যারে হিসেবে বিবেচনা করি, তাহলে তার প্রতিটি উপাদান হচ্ছে একটি বই। লিনিয়ার সার্চ পদ্ধতিতে আমরা লুপের সাহায্যে অ্যারের শুরু থেকে খুঁজতে থাকব এবং প্রতিটি উপাদান পরীক্ষা করে দেখব যে, সেটি আমাদের কাঙ্ক্ষিত উপাদান কি না। যদি আমরা আমাদের কাঙ্ক্ষিত উপাদানটি পেয়ে যাই, তাহলে খোঁজা বন্ধ করে লুপ থেকে বের হয়ে ঘোষণা করে দেব যে, আমরা যেটি খুঁজছিলাম সেটি পেয়ে গিয়েছি। আর যদি খুঁজতে খুঁজতে একেবারে লুপের শেষ পর্যন্ত এসেও সেটি খুঁজে না পাই, তাহলে ঘোষণা করব, এই অ্যারেতে এমন কোনো উপাদান নেই।



ছবি 3.1

লিনিয়ার সার্চ অ্যালগরিদম

ধাপ 1 : ধরি, অ্যারে-A এর উপাদান সংখ্যা n।

ধাপ 2 : `i=0` নেই।

ধাপ 3 : `i`-এর মান কি n-এর চেয়ে ছোটো? তাহলে ধাপ 4-এ যাই। নাহলে ধাপ 7-এ যাই।

ধাপ 4 : `A[i]` এবং x কি পরস্পর সমান? যদি হয়, তবে ধাপ 5-এ যাই।

ধাপ 5 : i রিটার্ন করি। (ফলাফল : i অবস্থানে কাঙ্ক্ষিত উপাদান খুঁজে পাওয়া গিয়েছে)

ধাপ 6 : i-এর মান 1 বাড়াই ও ধাপ 3-এ যাই।

ধাপ 7 : i=-1 ধরি ও i রিটার্ন করি। (ফলাফল : কাঙ্ক্ষিত উপাদান খুঁজে পাওয়া যায়নি)

অ্যালগরিদম লেখার কোনো বাঁধাধরা নিয়ম নেই, একটি অ্যালগরিদম বিভিন্নভাবে লেখা যায়। অ্যালগরিদম দেখে যদি বুঝতে পারা যায় যে, কোন কাজের পরে কোন কাজ করতে হবে, তাহলেই সেই অ্যালগরিদম সঠিক। তাই বিভিন্ন বইতে বিভিন্ন ধরন দেখে বিচলিত হওয়া যাবে না, যেকোনো একটি অনুসরণ করলেই হবে।

অ্যালগরিদম কখনো প্রোগ্রামিং ভাষার ওপর নির্ভর করে না, যেকোনো প্রোগ্রামিং ভাষাতেই অ্যালগরিদম ইমপ্লিমেন্ট করা যায়। আমরা যদি ওপরের অ্যালগরিদমটি একটি সি ফাংশন আকারে লিখি, তাহলে আমাদের কোড দাঁড়াবে এরকম–

```
int linear_search(int A[], int n, int x)
{
    int i;

    for (i = 0; i < n; i++) {
        if (A[i] == x) {
            return i; // ফলাফল : i  অবস্থানে কাঙ্ক্ষিত উপাদান খুঁজে
                      // পাওয়া গিয়েছে
        }
    }

    i = -1; // ফলাফল : কাঙ্ক্ষিত উপাদান খুঁজে পাওয়া যায়নি
    return i;
}
```

এখন, প্রশ্ন হচ্ছে, লিনিয়ার সার্চের স্পেস কমপ্লেক্সিটি কত? স্পেস কমপ্লেক্সিটি হচ্ছে $O(1)$। কারণ, প্রোগ্রামটি যতগুলো সংখ্যার ভেতরে খুঁজতে হবে, ততগুলো সংখ্যার তালিকা ইনপুট হিসেবে একটি অ্যারেতে থাকবে, তাই সেটি মূল অ্যালগরিদমের অংশ নয়। আর মূল অ্যালগরিদম একাধিক ভ্যারিয়েবল ব্যবহার করে, আর n-এর মান যতই হোক না কেন, সব ক্ষেত্রে কেবল ওই ভ্যারিয়েবলগুলোই ব্যবহার করা হবে। আর টাইম কমপ্লেক্সিটি হচ্ছে $O(n)$। কারণ nসংখ্যক উপাদানের জন্য আমাদেরকে সর্বোচ্চ nসংখ্যক বার তুলনা করে দেখতে হবে যে, আমরা কাঙ্ক্ষিত উপাদানটি পেলাম কি না। আবার অ্যালগরিদমটির বেস্ট কেইস (best case) কমপ্লেক্সিটি হচ্ছে $O(1)$, যদি লিস্টের প্রথম অবস্থানেই আমাদের কাঙ্ক্ষিত উপাদানটি থাকে। ওয়ার্স্ট কেইস

(worst case) কমপ্লেক্সিটি হচ্ছে $O(n)$, কারণ সর্বোচ্চ n সংখ্যক বার তুলনা লাগবে যদি সেটি লিস্টের শেষ অবস্থানে থাকে। worst শব্দের অর্থ সবচেয়ে খারাপ বা এক্ষেত্রে প্রতিকুল। আর এভারেজ কেস (average case) কমপ্লেক্সিটি হচ্ছে $O(\frac{n}{2})$, বা $O(n \times \frac{1}{2})$ বা $O(n) \times O(\frac{1}{2})$ বা $O(n)$। তবে বিভিন্ন অ্যালগরিদমের কমপ্লেক্সিটির কথা বলার সময় সাধারণত ওয়ার্স্ট কেস কমপ্লেক্সিটিই বোঝানো হয়।

বাইনারি সার্চ (Binary Search)

ধরা যাক, আমাকে অনেকগুলো সংখ্যার তালিকা দিয়ে বলা হলো সেখান থেকে একটি নির্দিষ্ট সংখ্যা (ধরি m) খুঁজে বের করতে। আর সেই সঙ্গে এই তথ্যও দিয়ে দেওয়া হলো যে, সংখ্যাগুলো ছোটো থেকে বড়ো ক্রমে সাজানো আছে। তাহলে আমি তালিকার সবচেয়ে মাঝামাঝিতে যেই সংখ্যাটি আছে, সেই সংখ্যাটি নেব। মাঝামাঝিতে দুটি সংখ্যা থাকলে, যেকোনো একটি (সাধারণত প্রথমটি) নেব। তারপরে সেই সংখ্যার সঙ্গে m-কে তুলনা করে দেখব– তারা কি পরস্পর সমান, নাকি ছোটো-বড়ো? যদি সমান হয়, তাহলে তো আমি আমার কাঙ্ক্ষিত সংখ্যাটি পেয়ে গেলাম। আর সমান না হলে, m তালিকার মধ্যের সংখ্যাটির চেয়ে বড়ো হতে পারে কিংবা ছোটো। যদি বড়ো হয়, সে ক্ষেত্রে আমি নিশ্চিত যে, আমাকে যে তালিকা দেওয়া হয়েছিল, সেই তালিকার প্রথম অর্ধেক সংখ্যাগুলোর মধ্যে m-কে পাওয়া যাবে না। কারণ, তালিকার প্রথম অর্ধেকের সংখ্যাগুলো যেহেতু মধ্যের সংখ্যাটির চেয়ে ছোটো, সেহেতু, তারা m-এর চেয়েও ছোটো হবে। আর m যদি মধ্যের সংখ্যাটির চেয়ে ছোটো হয়, তাহলে আমি বলতে পারি, তালিকার শেষ অর্ধেকের সংখ্যাগুলোর মধ্যে m নেই। কারণ, তালিকার শেষ অর্ধেকের সংখ্যাগুলো মধ্যের সংখ্যাটির চেয়ে বড়ো, সুতরাং তারা m-এর চেয়েও বড়ো। এবারে, দুই ভাগের মধ্যে যে ভাগে m আছে, সেই ভাগের মাঝামাঝি সংখ্যাটি আমরা নেব। তারপরে একইভাবে m-এর সঙ্গে তুলনা করব—তারা পরস্পর সমান, নাকি ছোটো-বড়ো? এভাবে প্রতিবার আমি যেই সংখ্যাগুলোর মধ্যে m-কে খুঁজছি, সেগুলোর প্রথম অর্ধেক বা শেষ অর্ধেক বাদ দিয়ে দেব।

এভাবে চলতে থাকলে একসময় m-কে খুঁজে পাওয়া যাবে। আর m-যদি সেই সংখ্যাগুলোর মধ্যে না থাকে, সেটি আমরা কীভাবে বুঝব? অর্ধেক বাদ দিতে দিতে একসময় আমাদের আর কোনো সংখ্যাই অবশিষ্ট থাকবে না। যদি কোনো সংখ্যাই আর অবশিষ্ট না থাকে, তাহলে আমরা বলে দিতে পারি যে, তালিকাটির মধ্যে m নেই। এই পদ্ধতিটিই হচ্ছে বাইনারি সার্চ। এখন আমরা দুটি উদাহরণ দেখব।

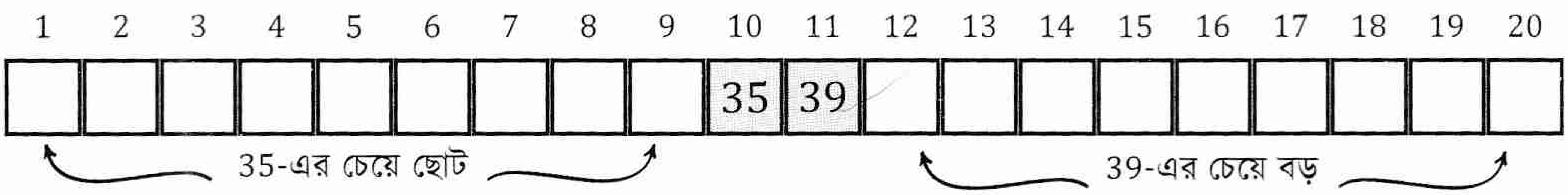
উদাহরণ ১ :

20টি কার্ড দেওয়া আছে। প্রতিটি কার্ডে একটি করে সংখ্যা লেখা আছে। কার্ডগুলো উপুড় করে রাখা আছে এবং সংখ্যার মান অনুযায়ী ছোটো থেকে বড়ো ক্রমে সাজানো আছে : 1, 4, 6, 7, 10, 19, 22, 23, 30, 35, 39, 46, 49, 50, 52, 55, 61, 67, 70, 71 (ছবি 3.2)। এই তালিকা থেকে আমাদের m=4 লেখা কার্ডটি খুঁজে বের করতে হবে, বাইনারি সার্চ পদ্ধতি প্রয়োগ করে।

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	4	6	7	10	19	22	23	30	35	39	46	49	50	52	55	61	67	70	71

ছবি 3.2

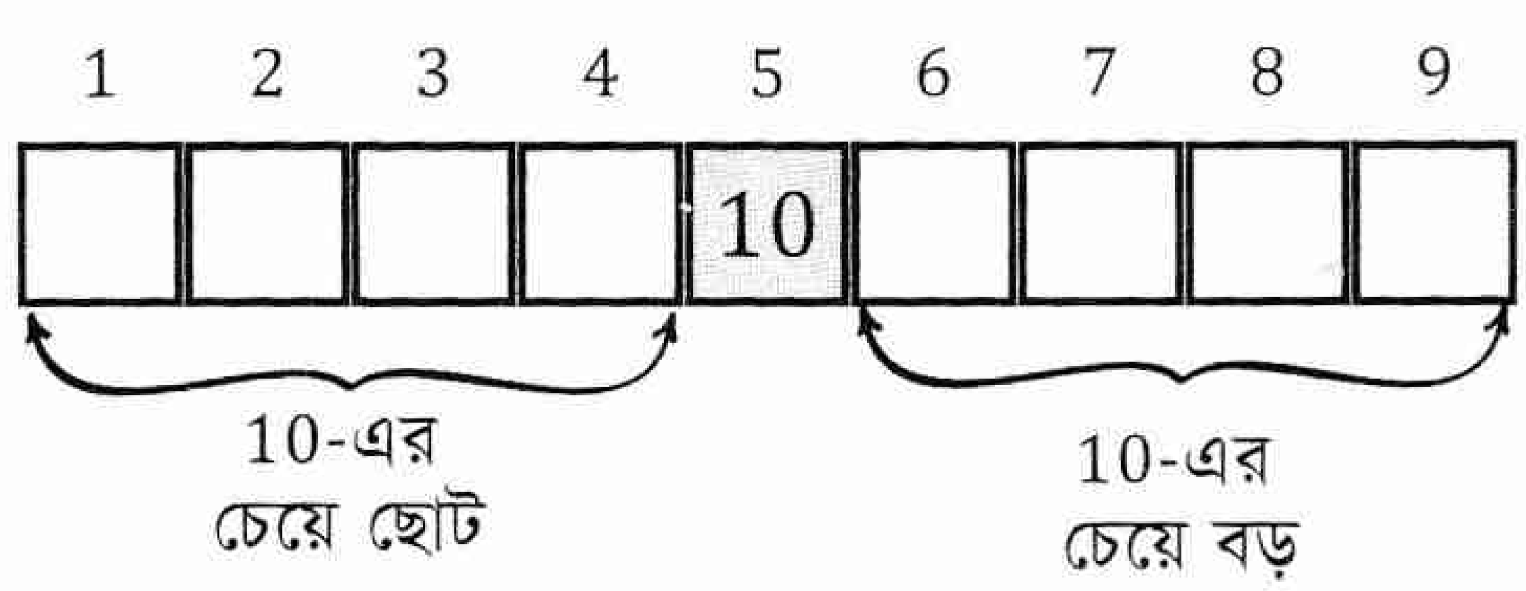
ধাপ 1 : কার্ডগুলোর মাঝামাঝি কার্ড হচ্ছে 10 ও 11 নম্বর কার্ড। আমরা প্রথমটি নেব, অর্থাৎ 10 নম্বর কার্ড যার মান 35 (ছবি 3.3)।



ছবি 3.3

এখন দেখি, 4 কি 35-এর সমান, নাকি ছোটো, নাকি বড়ো? উত্তর হচ্ছে, ছোটো। তাহলে, তালিকা থেকে 10 নম্বর ও তার পরের কার্ডগুলো আমরা বাদ দিয়ে দেব।

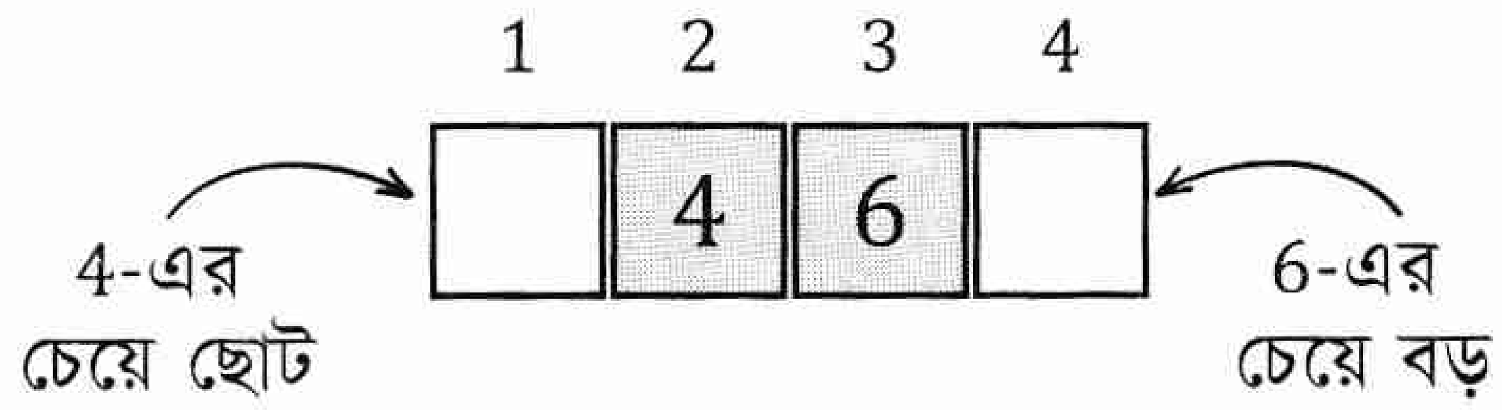
ধাপ 2 : অবশিষ্ট 9টি কার্ডের মধ্যের কার্ডটি হচ্ছে 5 নম্বর কার্ড, যার মান 10 (ছবি 3.4)।



ছবি 3.4

এখন দেখি, 4 কি 10-এর সমান, নাকি ছোটো, নাকি বড়ো? উত্তর হচ্ছে, ছোটো। তাহলে তালিকা থেকে 5 নম্বর ও তার পরের কার্ডগুলো আমরা বাদ দিয়ে দেব।

ধাপ 3 : অবশিষ্ট 4টি কার্ডের মধ্যের কার্ড হচ্ছে 2 ও 3 নম্বর কার্ড। আমরা প্রথমটি নেব, অর্থাৎ 2 নম্বর কার্ড যার মান 4 (ছবি 3.5)।



ছবি 3.5

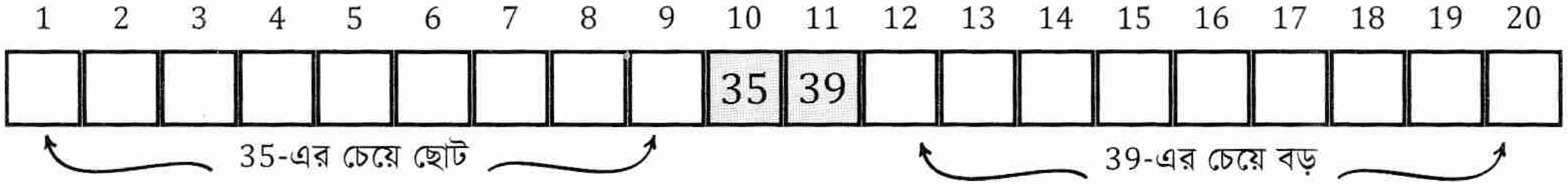
এখন দেখি, 4 কি 4-এর সমান, নাকি ছোটো, নাকি বড়ো? উত্তর হচ্ছে সমান। তাহলে আমরা আমাদের কাঙ্ক্ষিত সংখ্যাটি পেয়ে গিয়েছি।

তালিকায় 20টি কার্ড থাকলে মাত্র তৃতীয়বারের চেষ্টায় আমরা 4 লেখা কার্ডটিকে পেয়ে গেলাম!

উদাহরণ ২ :

ওপরের তালিকা থেকে আমাদেরকে যদি m=68 লেখা কার্ডটি খুঁজে বের করতে বলা হয়, তাহলে আমরা কী করব?

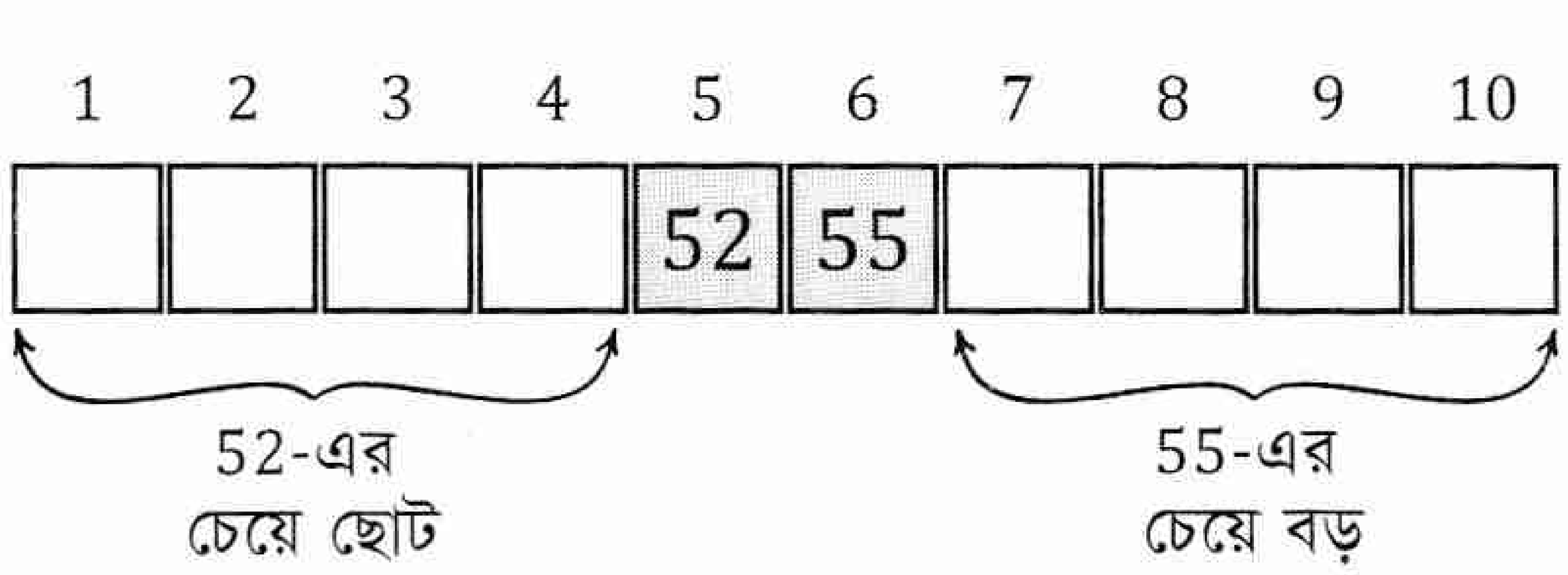
ধাপ 1 :কার্ডগুলোর মাঝমাঝি কার্ড হচ্ছে 10 ও 11 নম্বর কার্ড। আমরা প্রথমটি নেব, অর্থাৎ 10 নম্বর কার্ড যার মান 35 (ছবি 3.6)।



ছবি 3.6

এখন দেখি, 68 কি 35-এর সমান, নাকি ছোটো নাকি বড়ো? উত্তর হচ্ছে, বড়ো। তাহলে তালিকা থেকে 10 নম্বর ও তার আগের কার্ডগুলো আমরা বাদ দিয়ে দেব।

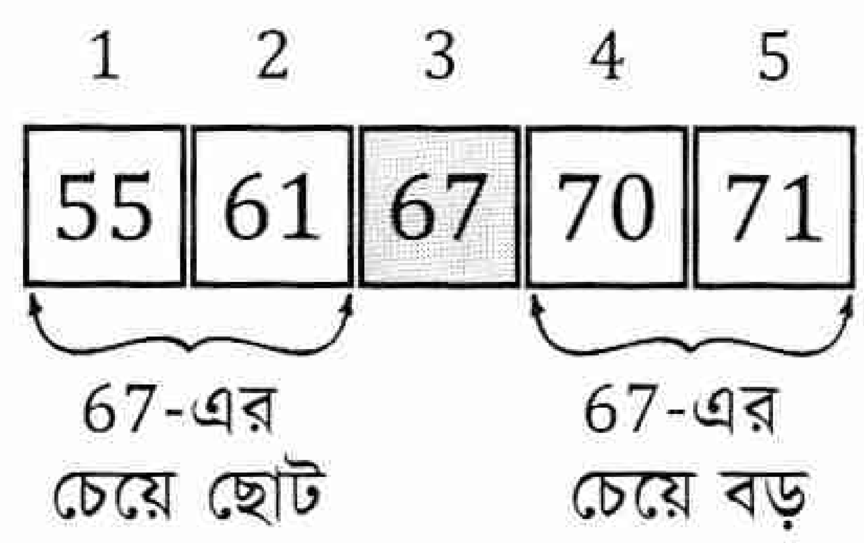
ধাপ 2 : অবশিষ্ট 10টি কার্ডের মাঝমাঝি কার্ড হচ্ছে 5 ও 6 নম্বর কার্ড। আমরা বরাবরের মতোই প্রথমটি নেব, অর্থাৎ 5 নম্বর কার্ড, যার মান 52 (ছবি 3.7)।



ছবি 3.7

এখন দেখি, 68 কি 52-এর সমান, নাকি ছোটো নাকি বড়ো? উত্তর হচ্ছে, বড়ো। তাহলে তালিকা থেকে 5 নম্বর ও তার আগের কার্ডগুলো আমরা বাদ দিয়ে দেব।

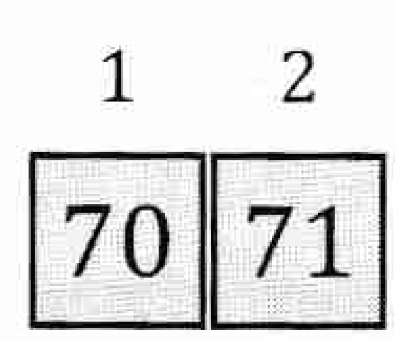
ধাপ 3 : অবশিষ্ট 5টি কার্ডের মধ্যের কার্ড হচ্ছে 3 নম্বর কার্ড যার মান 67 (ছবি 3.8)।



ছবি 3.8

67 যেহেতু 68-এর চেয়ে ছোটো, তাই আমরা 3 নম্বর ও তার আগের কার্ডগুলো বাদ দিয়ে দেব।

ধাপ 4 : অবশিষ্ট 2টি কার্ডের মধ্যের কার্ড হচ্ছে 1 ও 2 নম্বর কার্ড। তাই আমরা প্রথমটি নেব, অর্থাৎ 1 নম্বর কার্ড যার মান 70 (ছবি 3.9)।



ছবি 3.9

70 যেহেতু 68-এর চেয়ে বড়ো, তাই আমরা তালিকা থেকে 1 নম্বর ও তার পরের কার্ডগুলো বাদ দিয়ে দেব।

তাহলে, আমার তালিকাটি এখন খালি হয়ে গেল, অর্থাৎ আর কোনো কার্ড অবশিষ্ট নেই। তাই আমি বলে দিতে পারি, তালিকার মধ্যে 68 সংখ্যা লেখা কোনো কার্ড নেই।

এখন অ্যালগরিদমটি একটু সুন্দরভাবে লেখা যাক।

বাইনারি সার্চ অ্যালগরিদম

ইনপুট : একটি অ্যারে A এবং একটি উপাদান x।

আউটপুট : অ্যারের ইনডেক্স `i`, যেখানে সেই উপাদানটি খুঁজে পাওয়া গিয়েছে। খুঁজে না পেলে `i`-এর মান হবে -1।

ধাপ 1 : ধরি, A-এর উপাদান সংখ্যা হচ্ছে n।

ধাপ 2 : `left = 0` ও `right = n-1` নেই। আমরা অ্যারের `left` থেকে `right` ইনডেক্স পর্যন্ত সংখ্যাগুলোর মধ্যে উপাদানটি খুঁজব। `left` ও `right`-এর মান সমান হলে বুঝব অ্যারেতে একটিমাত্র উপাদান আছে। আর `left` যদি

`right`-এর চেয়ে বড়ো হয়, তার মানে অ্যারেতে আর কোনো উপাদান নেই বা থাকা সম্ভব নয়।

ধাপ 3 : যদি `left > right` হয়, তাহলে ধাপ 8 (আট)-এ যাই।

ধাপ 4 : `mid = (left + right) / 2` নেই। `mid` হচ্ছে `left` ও `right`-এর মাঝামাঝি ইনডেক্স।

ধাপ 5 : `L[mid] == x` হলে, `mid` রিটার্ন করি। (ফলাফল : কাঙ্ক্ষিত উপাদানটি `mid` অবস্থানে খুঁজে পাওয়া গিয়েছে।)

ধাপ 6 : `L[mid] < x` হলে, `left = mid + 1` নেই। তারপরে ধাপ 3-এ যাই।

ধাপ 7 : `L[mid] > x` হলে, `right = mid - 1` নেই। তারপরে ধাপ 3-এ যাই।

ধাপ 8 : কাঙ্ক্ষিত উপাদান খুঁজে পাওয়া যায়নি, তাই `-1` রিটার্ন করি।

বাইনারি সার্চের স্পেস কমপ্লেক্সিটি লিনিয়ার সার্চের মতোই, অর্থাৎ $O(1)$। কারণ, এখানে আমরা কয়েকটি ভ্যারিয়েবল ব্যতীত অতিরিক্ত কোনো মেমোরি ব্যবহার করি না আর ভ্যারিয়েবলের সংখ্যা ধ্রুবক (`n, left, right, mid`—এই চারটি)। আর অ্যারে (`A`) মূল অ্যালগরিদমের অংশ নয়, বরং ইনপুট। বাইনারি সার্চের টাইম কমপ্লেক্সিটি হচ্ছে $O(\log n)$। এতক্ষণে পাঠক একটু নড়েচড়ে বসতে পারে। এ আবার কেমন কমপ্লেক্সিটি?

স্কুলের গণিত বইতে লগারিদম যদি আমাদের মনে থাকে (মনে না থাকলে নবম-দশম শ্রেণির বই জোগাড় করে দেখে নিতে হবে), তাহলে আমরা জানি, $b^y = x$ হলে, $\log_b x = y$। তাহলে $\log_2 16 = 4$, কারণ $2^4 = 16$। 2^6 হচ্ছে $2 \times 2 \times 2 \times 2 \times 2 \times 2 = 64$। তাহলে $\log_2 64$ হচ্ছে 6। অতএব, আমরা বলতে পারি, ঘাত বা সূচক (ইংরেজিতে পাওয়ার—power)-এর উলটো অপারেশনটি হচ্ছে লগ। 1-কে 5-বার 2 দিয়ে গুণ করলে আমরা 32 পাই ($1 \times 2 \times 2 \times 2 \times 2 \times 2$)। তাহলে 32-কে 2 দিয়ে ভাগ করতে থাকলে (অর্থাৎ অর্ধেক করতে থাকলে), আমরা 5 বার ভাগ করার পরে 1 পাব।

$$\frac{32}{2} = 16$$

$$\frac{16}{2} = 8$$

$$\frac{8}{2} = 4$$

$$\frac{4}{2} = 2$$

$$\frac{2}{2} = 1$$

তাহলে, $\log_2 32$-এর মান হবে 5। বাইনারি সার্চে যেহেতু আমি প্রতিবার আমার লিস্টকে অর্ধেক করে ফেলছি, তাই সর্বোচ্চ $\log_2 n$সংখ্যক বার আমরা কাজটি করতে পারব। এর চেয়ে একবার বেশি ভাগ করলে ভাগফল হবে শূন্য, অর্থাৎ লিস্টে আর কিছু নেই। আর প্রতিবারই আমরা তিনটি কম্পারিজন বা তুলনা করছি (সমান, নাকি ছোটো, নাকি বড়ো?)। আর কম্পিউটার বিজ্ঞানে $\log_2 n$-কে লেখা হয় $\log n$। অর্থাৎ, লগ-এর ভিত্তি বা বেইজ (base) উল্লেখ করা না থাকলে, ধরে নেওয়া হয় সেটি 2-ভিত্তিক লগ। তাই বাইনারি সার্চের কমপ্লেক্সিটি আমরা লিখি $O(\log n)$।

এখন প্রশ্ন হচ্ছে, এই $O(\log n)$ কি $O(n)$-এর চেয়ে ভালো, নাকি খারাপ? লিনিয়ার সার্চের ক্ষেত্রে, অ্যারেতে 1 কোটি উপাদান থাকলে, সর্বোচ্চ 1 কোটিবার লুপ চলবে এবং প্রতিবার একটি করে তুলনা করা হবে। আর বাইনারি সার্চের ক্ষেত্রে 1 কোটি উপাদান থাকলে লুপটি চলবে সর্বোচ্চ 24 বার মাত্র। তাহলে আমরা দেখতে পাচ্ছি, অ্যালগরিদমের কমপ্লেক্সিটি যদি $O(\log n)$হয়, তাহলে সেটি বেশ ভালো, অর্থাৎ কম কাজেই ফল পাওয়া যায়।

এখন আমি বাইনারি সার্চের জন্য কোড লিখব—

```
int binary_search(int A[], int n, int x)
{
    int left, right, mid;
    left = 0;
    right = n - 1;

    while (left <= right) {
        mid = (left + right) / 2;

        if (A[mid] == x) {
            return mid;    // কাঙ্ক্ষিত উপাদান খুঁজে পাওয়া গিয়েছে
        }

        if (A[mid] < x) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
```

```
    return -1;     // কাঙ্ক্ষিত উপাদান খুঁজে পাওয়া যায়নি
}
```

তোমাদের কাজ হচ্ছে, আরো কিছু কোড লিখে binary_search() ফাংশনটি পরীক্ষা করে দেখা যে, সেটি ঠিকঠাক কাজ করছে কি না। এজন্য তোমরা একটি লুপের সাহায্যে 0 থেকে 100 উপাদানের অ্যারে তৈরি করবে, এবং তারপর প্রতিটি উপাদান খুঁজে দেখবে যে সঠিক স্থানে পাওয়া যায় কি না। আবার যেসব উপাদান নেই, সেসব উপাদানের জন্যও ফাংশনটি –1 রিটার্ন করে কি না, তাও পরীক্ষা করে দেখতে হবে।

উল্লেখ্য যে, অনেক সময় left ও right-এর মান যদি এমন হয় যে, তাদের যোগফল $2^{31} - 1$ (অর্থাৎ 2147483647)-এর চেয়ে বেশি হয়, তখন ইন্টিজার ওভারফ্লো হতে পারে। তাই mid = (left + right)/2 এর পরিবর্তে, mid = left + (right - left) / 2 লেখা হয়, যেন ইন্টিজার ওভারফ্লো সমস্যার সৃষ্টি না হয়। তোমরা নিজেরা বিষয়টি একটু চিন্তা করে দেখতে পারো।

অধ্যায় ৪ : সর্টিং অ্যালগরিদম (Sorting Algorithm)

সর্ট (sort) করা মানে একটি নির্দিষ্ট ক্রমে সাজানো। যেমন, অভিধান বা ডিকশনারিতে শব্দগুলো বর্ণমালার ক্রমানুসারে সাজানো থাকে। আবার শ্রেণিকক্ষে হাজিরা খাতায় নামগুলো রোল নম্বরের ছোটো থেকে বড়ো ক্রমে সাজানো থাকে। আবার একটি ক্রিকেট টুর্নামেন্ট শেষে সর্বোচ্চ রান সংগ্রহকারী 10 জন ব্যাটসম্যানের নাম সাজানো থাকে মোট রানের বড়ো থেকে ছোটো ক্রমে। কম্পিউটারকে প্রায় সময়ই এই সাজানো বা সর্ট করার কাজটি করতে হয়। আর এই কাজের জন্য রয়েছে নানান রকম অ্যালগরিদম। এই অধ্যায়ে আমরা এরকম তিনটি খুব সহজ অ্যালগরিদম শিখব। অ্যালগরিদমগুলো হচ্ছে, সিলেকশন সর্ট (Selection sort), বাবল সর্ট (Bubble sort) ও ইনসারশন সর্ট (Insertion sort)।

সিলেকশন সর্ট (Selection sort)

সিলেক্ট (select) শব্দের অর্থ সবচেয়ে ভালো বা উপযোগীকে বাছাই করা। আমাকে যদি অনেকগুলো উপাদানের তালিকা দেওয়া হয়, আর সেখান থেকে আমি চিন্তাভাবনা ছাড়াই যেকোনো একটি উপাদান বেছে নিই, সেটি কিন্তু সিলেক্ট করা হবে না। বরং, আমি চিন্তাভাবনা করে সবচেয়ে দরকারি বা উপযোগী বা যোগ্য কিংবা ভালো উপাদানটি বাছাই করলে সেটিকে বলে সিলেক্ট করা। সিলেকশন সর্টের বেলাতে, প্রতিবার অ্যারে থেকে আমি সবচেয়ে উপযোগী উপাদানটি বেছে নেব এবং তারপর তাকে একটি নির্দিষ্ট জায়গায় রাখব। তারপর তালিকার অবশিষ্ট উপাদানগুলোর জন্যও একই পদ্ধতি প্রয়োগ করব। এভাবে বারবার করতে থাকলে একসময় তালিকার উপাদানগুলো সঠিক ক্রমে সাজানো হয়ে যাবে।

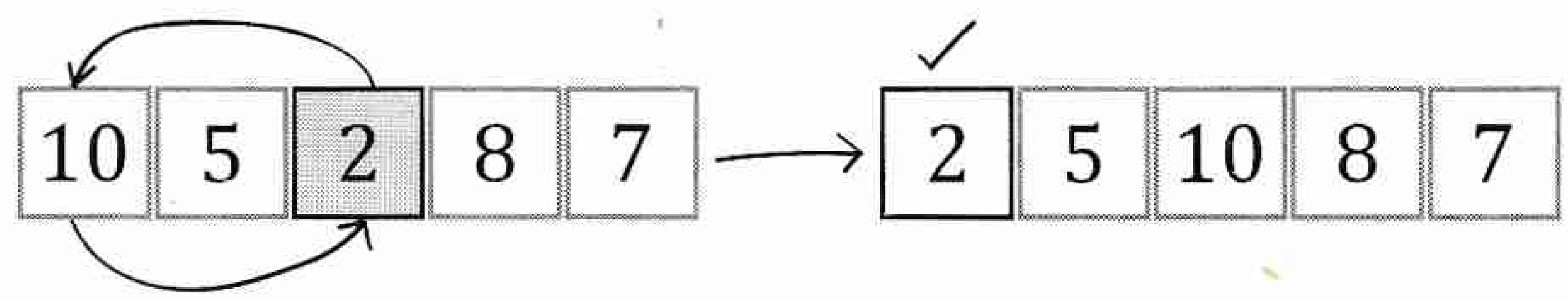
ধরা যাক, পাঁচটি সংখ্যা দিয়ে আমাদের বলা হলো, সংখ্যাগুলোকে ছোটো থেকে বড়ো ক্রমে সাজাতে হবে। সংখ্যাগুলো হচ্ছে– 10, 5, 2, 8, 7 (ছবি 4.1)। এই সাজানোর কাজটি করার জন্য আমরা সিলেকশন সর্ট পদ্ধতি ব্যবহার করব।

10	5	2	8	7

ছবি 4.1

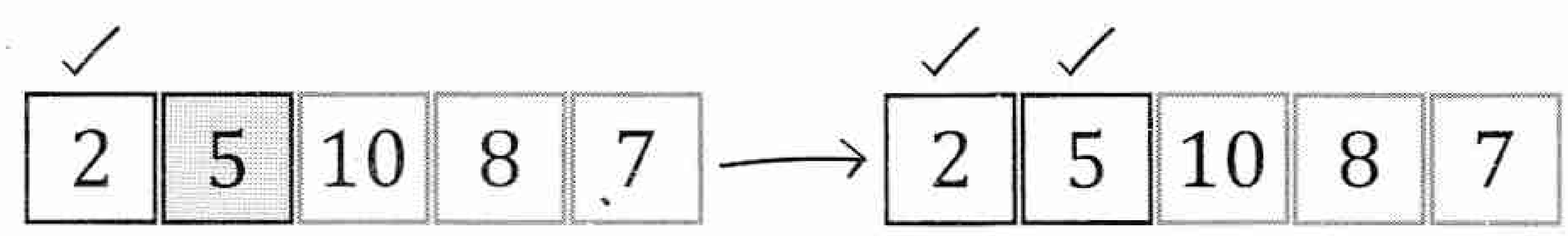
ধাপ 1 : প্রথমে পাঁচটি সংখ্যা থেকে সবচেয়ে ছোটো সংখ্যাটি বাছাই করতে হবে। এ ক্ষেত্রে সেই সংখ্যাটি হচ্ছে, 2। এখন 2-কে সবার প্রথমে নিয়ে আসতে হবে। তাই তালিকার প্রথমে

যে সংখ্যাটি আছে (10), তার সঙ্গে 2-এর স্থানের অদলবদল করতে হবে। তাহলে সংখ্যাগুলোর তালিকা দাঁড়াবে এরকম– 2, 5, 10, 8, 7 (ছবি 4.2)।



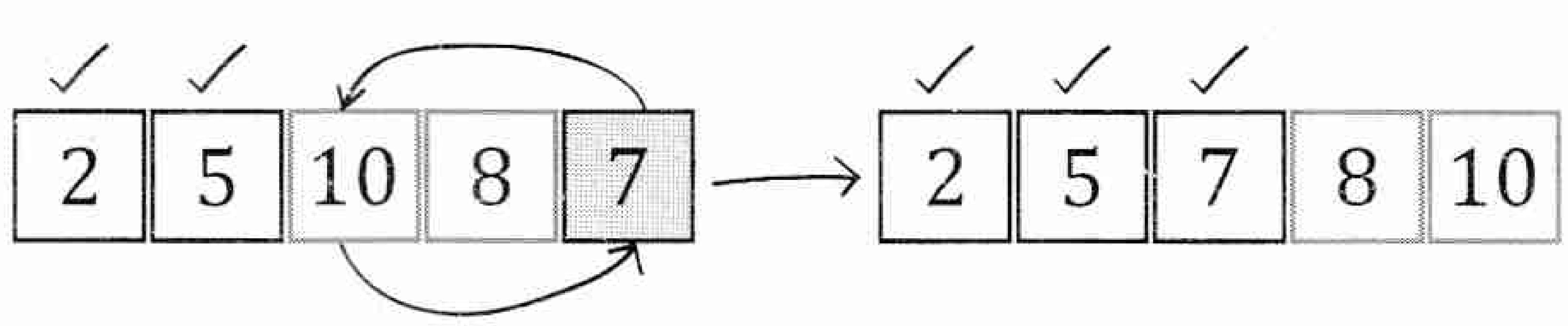
ছবি 4.2

ধাপ 2 :এখন আমরা জানি যে, তালিকার শুরুতে সবচেয়ে ছোটো সংখ্যাটি আছে। তাই বাকি চারটি সংখ্যার মধ্যে সবচেয়ে ছোটো সংখ্যাটি বাছাই করতে হবে, সেটি হচ্ছে 5। যেহেতু এই চারটি সংখ্যার মধ্যে 5 সবার শুরুতে আছে, তাই 5-কে আর অদলবদল করার প্রয়োজন নেই (ছবি 4.3)।



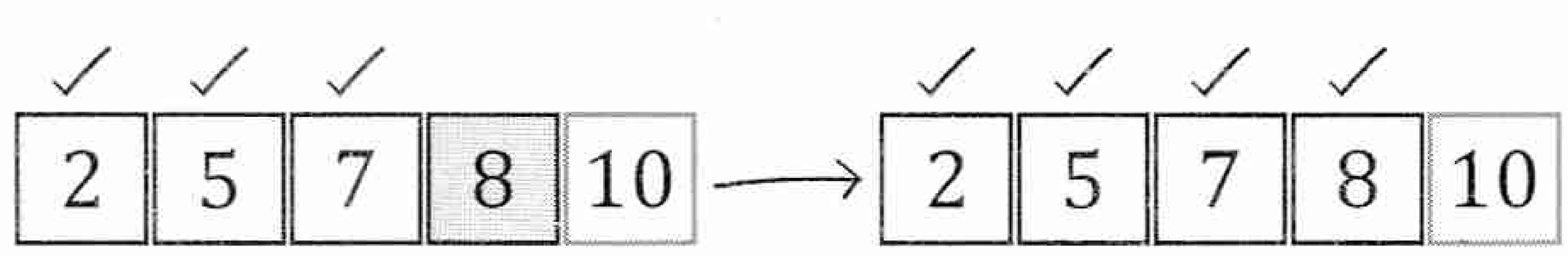
ছবি 4.3

ধাপ 3 : এবারে 10, 8, 7 – এই তিনটি সংখ্যার মধ্যে সবচেয়ে ছোটো সংখ্যাটি বাছাই করতে হবে। সেটি হচ্ছে 7। এবারে 7-কে এই তিনটি সংখ্যার মধ্যে সবচেয়ে প্রথমে রাখতে হবে, তাই 10-এর সঙ্গে স্থানের অদলবদল করতে হবে। এখন পুরো তালিকাটি দাঁড়াবে এরকম– 2, 5, 7, 8, 10 (ছবি 4.4)। তাহলে এতদূর আসার পরে আমরা জানি, প্রথম তিনটি স্থানের সংখ্যাগুলো ঠিকঠাকমতো আছে।



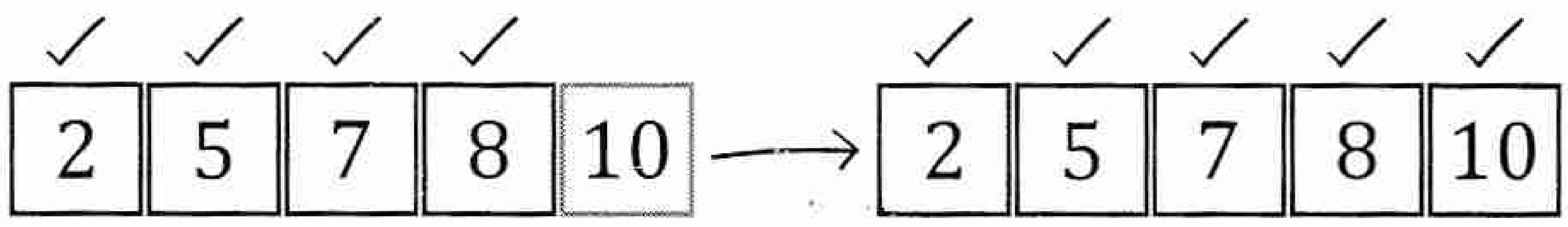
ছবি 4.4

ধাপ 4 : এখন বাকি দুটি সংখ্যা 8, 10-এর মধ্য থেকে সবচেয়ে ছোটো সংখ্যাটি বেছে, এদের মধ্যে প্রথম স্থানে রাখতে হবে। সেই সংখ্যাটি হচ্ছে 8। আর 8 যেহেতু শুরুতেই আছে, তাই আর স্থানের অদলবদল করতে হবে না। এই ধাপের শেষে আমরা নিশ্চিতভাবে বলতে পারি, প্রথম চারটি স্থানে সংখ্যাগুলো ছোটো থেকে বড়ো ক্রমে আছে (ছবি 4.5)।



ছবি 4.5

আবার যেহেতু, পাঁচটি সংখ্যার মধ্যে সবচেয়ে ছোটো চারটি সংখ্যা শুরুতে ছোটো থেকে বড়ো ক্রমানুসারে আছে, তাই পঞ্চম সংখ্যাটি (অর্থাৎ 10) তার যথাযথ অবস্থানে আছে (ছবি 4.6)। সেই সংখ্যাটি নিয়ে আমাদেরকে আর কিছু করতে হবে না।



ছবি 4.6

অনুশীলনী :

ছয়টি সংখ্যা দেওয়া আছে এরকম– 8, 2, 4, 1, 5, 7। সিলেকশন সর্ট পদ্ধতি ব্যবহার করে সংখ্যাগুলোকে ছোটো থেকে বড়ো ক্রমে সাজাতে হবে।

আশা করি, সিলেকশন সর্ট করার পদ্ধতি সবাই বুঝে ফেলেছে এবং ওপরের অনুশীলনী ঠিকভাবে উত্তর করতে পেরেছে। এখন সিলেকশন সর্টের অ্যালগরিদমটি লিখে ফেলা যাক।

সিলেকশন সর্ট অ্যালগরিদম

ইনপুট : একটি অ্যারে `A`, যেখানে nসংখ্যক উপাদান আছে এবং উপাদানগুলোর পরস্পরের মধ্যে তুলনা করে বড়ো-ছোটো নির্ধারণ করা যায়।

ধাপ 1 : `i` নামে একটি ভ্যারিয়েবল নিই, এবং এর মান `0` থেকে শুরু করে `n-1`-এর আগ পর্যন্ত এক-এক করে বাড়াই। `i`-এর প্রতিটি মানের জন্য ধাপ 2-এর কাজ করি।

ধাপ 2 : `A[i]` থেকে `A[n-1]` পর্যন্ত উপাদানগুলোর মধ্যে সবচেয়ে ছোটো উপাদানটির ইনডেক্স বের করে, সেই উপাদানকে `i`-তম ইনডেক্সে অবস্থিত উপাদানের সঙ্গে অদলবদল করি।

ধাপ 3 : অ্যারে `A`-তে সংখ্যাগুলো ছোটো থেকে বড়ো ক্রমে সাজানো আছে।

চাইলে অ্যালগরিদমটি আরো বিস্তারিত লেখা যায় নিচের মতো করে।

ধাপ 1 : ধরি, `i = 0`।

ধাপ 2 : i যদি `n-1`-এর চেয়ে বড়ো কিংবা সমান হয়, তাহলে ধাপ 11-এ যাই।

ধাপ 3 : ধরি, লিস্টের সবচেয়ে ছোটো উপাদানটির ইনডেক্স, `index_min = i`।

ধাপ 4 : ধরি, `j = i + 1`।

ধাপ 5 : j যদি n-এর চেয়ে বড়ো কিংবা সমান হয়, তাহলে ধাপ 9-এ যাই।

ধাপ 6 : যদি `A[j] < A[index_min]` হয়, তাহলে পরবর্তী ধাপে যাই। নইলে ধাপ 8 (আট)-এ যাই।

ধাপ 7 : `index_min = j`।

ধাপ 8 : j-এর মান এক বাড়াই (`j = j + 1`)। ধাপ 5-এ যাই।

ধাপ 9 : যদি i আর index_min সমান না হয়, তবে A[i] ও A[index_min] অদলবদল করি।

ধাপ 10 : i-এর মান এক বাড়াই (`i = i + 1`)। ধাপ 2-এ যাই।

ধাপ 11 : অ্যারে A-তে সংখ্যাগুলো ছোটো থেকে বড়ো ক্রমে সাজানো আছে।

ওপরের দুটি অ্যালগরিদমের শেষ ধাপটি না লিখলেও সমস্যা নেই। আমি লিখেছি যেন তোমরা বুঝতে পারো যে, এই পর্যায়ে এসে অ্যালগরিদমের কাজ শেষ।

এখন অ্যালগরিদমটি ইমপ্লিমেন্ট করার পালা।

```
void selection_sort(int A[], int n)
{
    int i, j, index_min, temp;

    for (i = 0; i < n-1; i++) {
        index_min = i;

        for (j = i+1; j < n; j++) {
            if (A[j] < A[index_min]) {
                index_min = j;
            }
        }

        if (index_min != i) {
            // A[i] ও A[index_min]-এর উপাদানগুলো অদলবদল
            // করতে হবে।

            temp = A[i];
```

```
            A[i] = A[index_min];
            A[index_min] = temp;
        }
    }
}
```

ওপরে আমি `selection_sort()` ফাংশনে সিলেকশন সর্ট অ্যালগরিদম ইমপ্লিমেন্ট করলাম। তোমরা এখন মেইন ফাংশনে একটি অ্যারে তৈরি করে, সেখানে থেকে `selection_sort()` ফাংশন কল করে, তারপর অ্যারেটি প্রিন্ট করে দেখো যে, ঠিকঠাকভাবে সর্ট হচ্ছে কি না।

এখন প্রশ্ন হচ্ছে, সিলেকশন সর্টের টাইম কমপ্লেক্সিটি কত? কোড ভালোভাবে খেয়াল করলে আমরা দেখব, প্রথম লুপটি চলে `0` থেকে `n-2` পর্যন্ত। আবার ভেতরের লুপটি চলে, `i+1` থেকে `n-1` পর্যন্ত। তার ভেতরে একটি তুলনা বা কম্পারিজন অপারেশন (`A[j] < A[index_min]`) এবং সেটি সত্যি হলে একটি অ্যাসাইনমেন্ট অপারেশন (`index_min = j`) করা হয়। তারপর লুপের বাইরেও একটি কম্পারিজন অপারেশন (`index_min != i`) আর সেটি সত্যি হলে, তিনটি অ্যাসাইনমেন্ট অপারেশন করা হয়, তবে সেটি হয় (`n-1`)সংখ্যক বার। তাই সেটি আমাদের বিবেচনা করার প্রয়োজন নেই। এখন দ্বিতীয় লুপের ভেতরের কাজটি কতবার হয় সেটি বের করতে হবে, কারণ সেটিই আসলে আমাদের প্রোগ্রামের বা অ্যালগরিদমের কমপ্লেক্সিটি নির্ধারণ করবে।

যখন, `i=0`, তখন ভেতরের লুপটি চলে `j=1` থেকে `j=n-1` পর্যন্ত, মোট (`n-1`)সংখ্যক বার।
যখন, `i=1`, তখন ভেতরের লুপটি চলে `j=2` থেকে `j=n-1` পর্যন্ত, মোট (`n-2`)সংখ্যক বার।
যখন, `i=2`, তখন ভেতরের লুপটি চলে `j=3` থেকে `j=n-1` পর্যন্ত, মোট (`n-3`)সংখ্যক বার।

এভাবে,

যখন, `i=n-3`, তখন ভেতরের লুপটি চলে `j=n-2` থেকে `j=n-1` পর্যন্ত, মোট 2সংখ্যক বার।
যখন, `i=n-2`, তখন ভেতরের লুপটি চলে `j=n-1` থেকে `j=n-1` পর্যন্ত, মোট 1সংখ্যক বার।

এবারে, সবগুলো যোগ করে দিই, `(n-1) + (n-2) + (n-3) + ... + 2 + 1`

এই ধারার যোগফলকে লেখা যায়, $\frac{n\,(n-1)}{2}$, বা $\frac{n^2}{2} - \frac{n}{2}$

তাই কমপ্লেক্সিটি হচ্ছে, $O(\frac{n^2}{2}) - O(\frac{n}{2})$

বা, $O(\frac{n^2}{2})$,

বা $O(n^2)$।

ওপরের গাণিতিক প্রমাণটুকু না বুঝলে নবম-দশম শ্রেণির গণিতের বইয়ের ধারার অংশটুকু পড়ে নিতে হবে। আর অভিজ্ঞতা একটু বাড়লে আমরা এমনিতেই বুঝতে পারব যে, একটি লুপ n-সংখ্যকবার চললে, এবং তার ভেতরে আরেকটি লুপ যদি mসংখ্যক বার চলে, তাহলে টাইম কমপ্লেক্সিটি হয় $O(n \times m)$। এখন m–এর মান n–এর সমান হলে, $O(n \times n)$ বা $O(n^2)$। এমনকি, m যদি $\frac{n}{2}$-এর সমান হয়, তাহলে $O(\frac{n \times n}{2})$ বা $O(\frac{n^2}{2})$ বা $O(n^2)$।

বাবল সর্ট (Bubble sort)

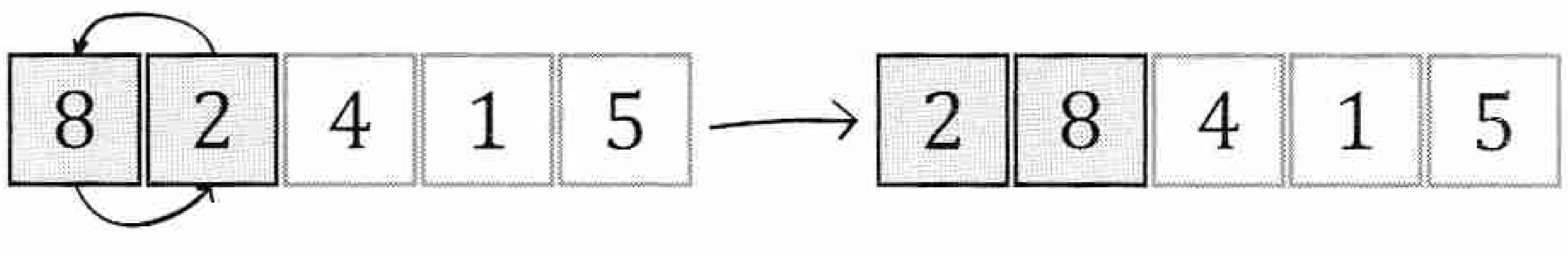
বাবল (bubble) শব্দের অর্থ হচ্ছে বুদ্বুদ। পানিতে যেমন বুদ্বুদ ওপরের দিকে ওঠে, তেমনি বাবল সর্টে পরপর দুটি জিনিস তুলনা করে তাদের মধ্যকার ক্রম অনুসারে রাখতে হবে আর এভাবে করতে থাকলে সবচেয়ে বড়ো জিনিসটি (যদি আমরা ছোটো থেকে বড়ো ক্রমে সাজাতে চাই), কিংবা সবচেয়ে ছোটো জিনিসটি (যদি আমরা বড়ো থেকে ছোটো ক্রমে সাজাতে চাই) অ্যারের শেষে চলে আসবে। এভাবে একটি একটি করে জিনিস বুদ্বুদের মতো উঠে তাদের নিজের জায়গায় বসে যাবে। এজন্যই একে বলা হয় বাবল সর্ট। একটি উদাহরণ দিলে বিষয়টি পরিষ্কার হবে।

পাঁচটি সংখ্যা দেওয়া আছে : 8, 2, 4, 1, 5 (ছবি 4.7)। এগুলোকে ছোটো থেকে বড়ো ক্রমে সাজাতে হবে।



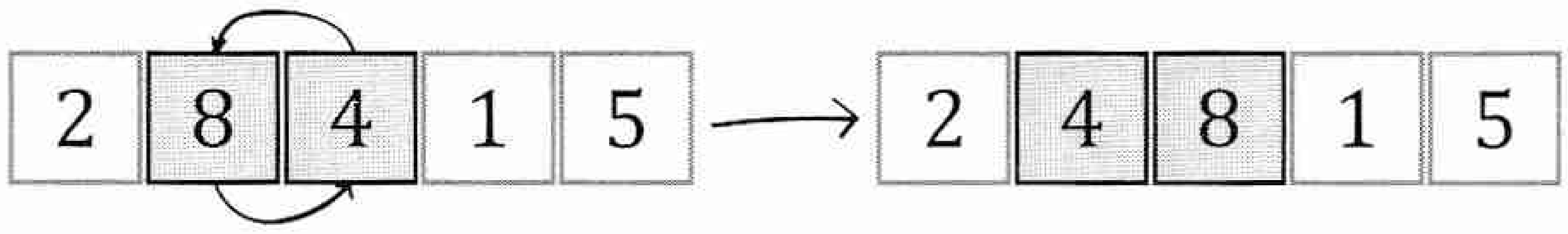
ছবি 4.7

ধাপ 1 : প্রথমে 8 ও 2 এর মধ্যে তুলনা করে তাদের মধ্যে ছোটো সংখ্যাটি প্রথম স্থানে, আর বড়ো সংখ্যাটি তার পরে রাখব। যেহেতু 8 বড়ো, তাই প্রথমে 2, তারপরে 8 রাখব। তাহলে সংখ্যাগুলোর তালিকা দাঁড়াবে এমন– 2, 8, 4, 1, 5 (ছবি 4.8)।



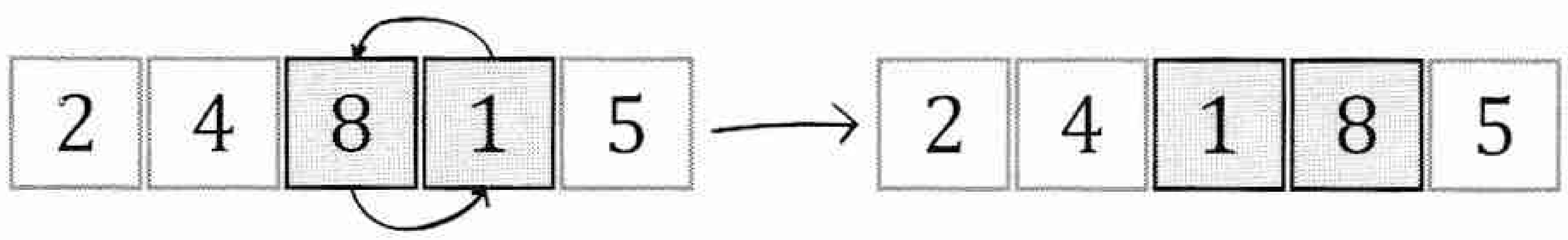
ছবি 4.8

ধাপ 2 : এবারে 8 ও 4-এর মধ্যে তুলনা করে ছোটোটি আগে ও বড়োটি পরে রাখব– 2, 4, 8, 1, 5 (ছবি 4.9)।



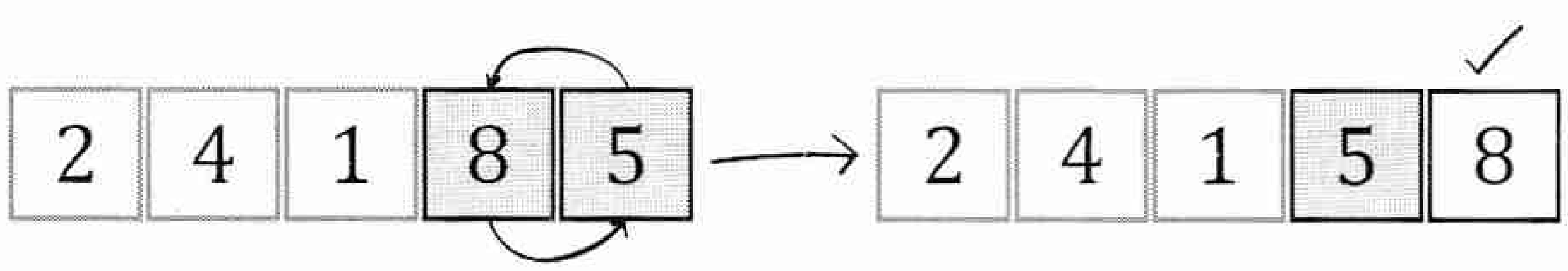
ছবি 4.9

ধাপ 3 : একই কাজ করা হবে 8 ও 1-এর মধ্যে– 2, 4, 1, 8, 5 (ছবি 4.10)।



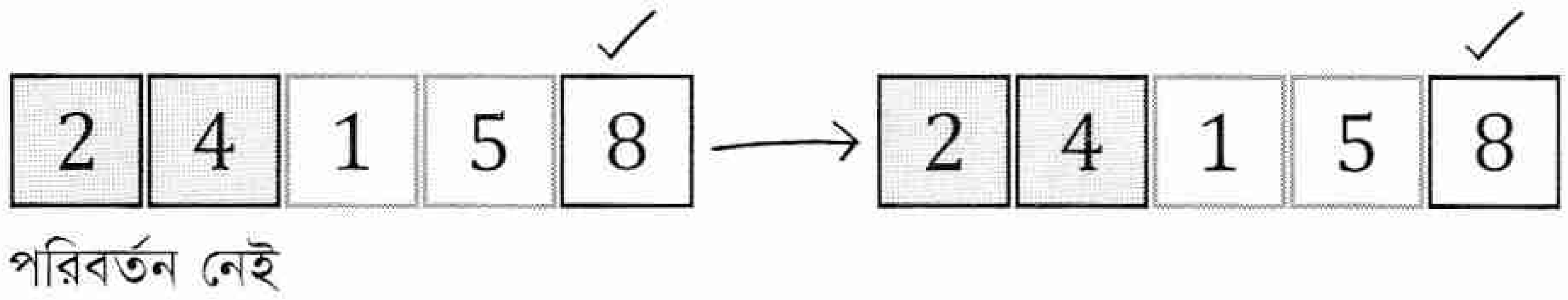
ছবি 4.10

ধাপ 4 : এবারে 8 ও 5-এর মধ্যে তুলনা করে বড়োটি শেষে রাখব– 2, 4, 1, 5, 8। এই ধাপের শেষে তালিকার সবচেয়ে বড়ো সংখ্যাটি সবার শেষে চলে আসল (ছবি 4.11)। এখন প্রথম চারটি সংখ্যার মধ্যে একই কাজ করতে হবে।



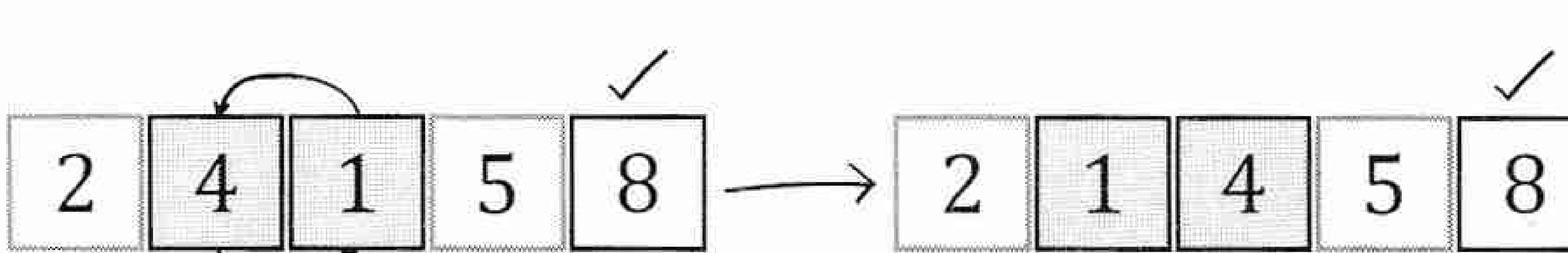
ছবি 4.11

ধাপ 5 : 2 ও 4 তুলনা করি। যেহেতু 2 ছোটো, 4 বড়ো, তাই তাদের মধ্যে স্থানের অদলবদল করতে হবে না (ছবি 4.12)।



ছবি 4.12

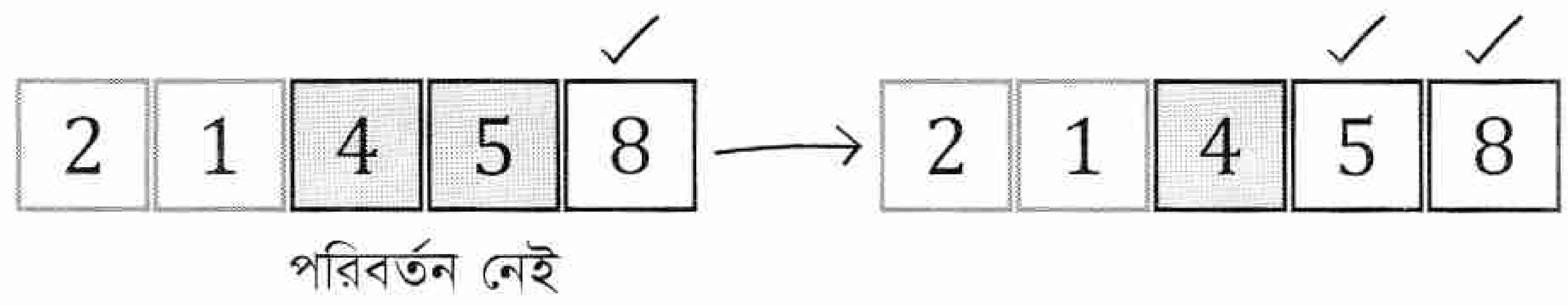
ধাপ 6 : 4 ও 1 তুলনা করি। যেহেতু 1 ছোটো, 4 বড়ো, তাই প্রথমে 1 রাখতে হবে, তারপরে 4 রাখতে হবে। তালিকাটি দাঁড়াচ্ছে এমন– 2, 1, 4, 5, 8 (ছবি 4.13)।



ছবি 4.13

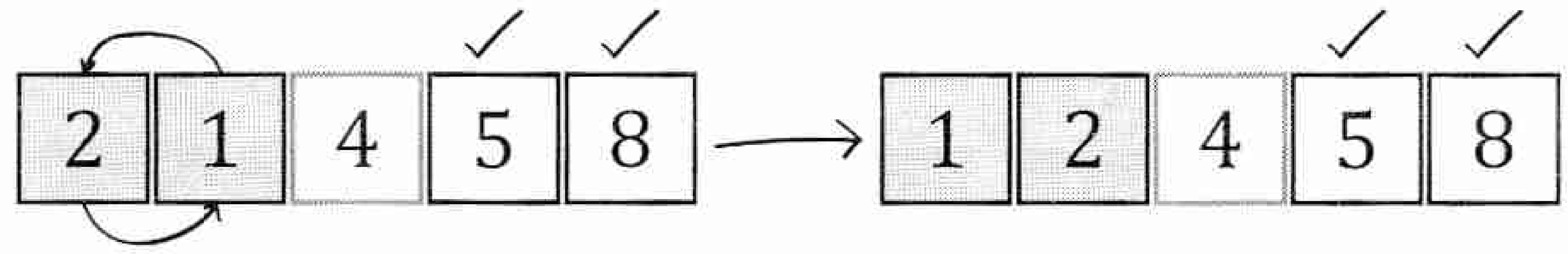
ধাপ 7 : 4 ও 5 তুলনা করি। এখানে তাদের স্থান বদলাতে হবে না, কারণ 4 ছোটো 5 বড়ো। এই ধাপের শেষে আমরা নিশ্চিত হলাম যে, সবচেয়ে বড়ো দুটি সংখ্যা সবচেয়ে শেষে আছে

(5, 8) (ছবি 4.14)। এখন প্রথম তিনটি সংখ্যার মধ্যে আমরা ওপরের কাজের পুনরাবৃত্তি করব।



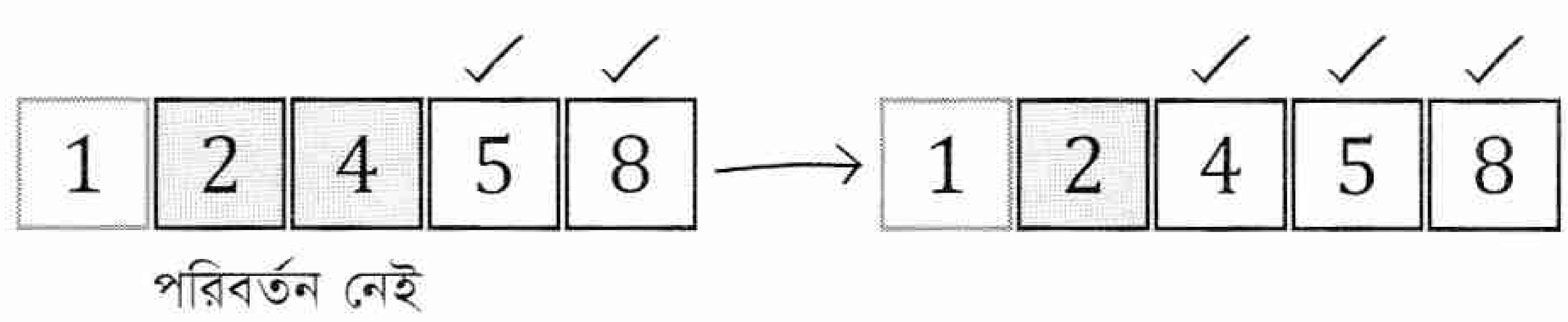
ছবি 4.14

ধাপ 8 : 2 ও 1 তুলনা করি। যেহেতু 2 বড়ো 1 ছোটো, তাই প্রথমে 1 রাখতে হবে, তারপর 2। তালিকাটি এখন এরকম হবে– 1, 2, 4, 5, 8 (ছবি 4.15)।



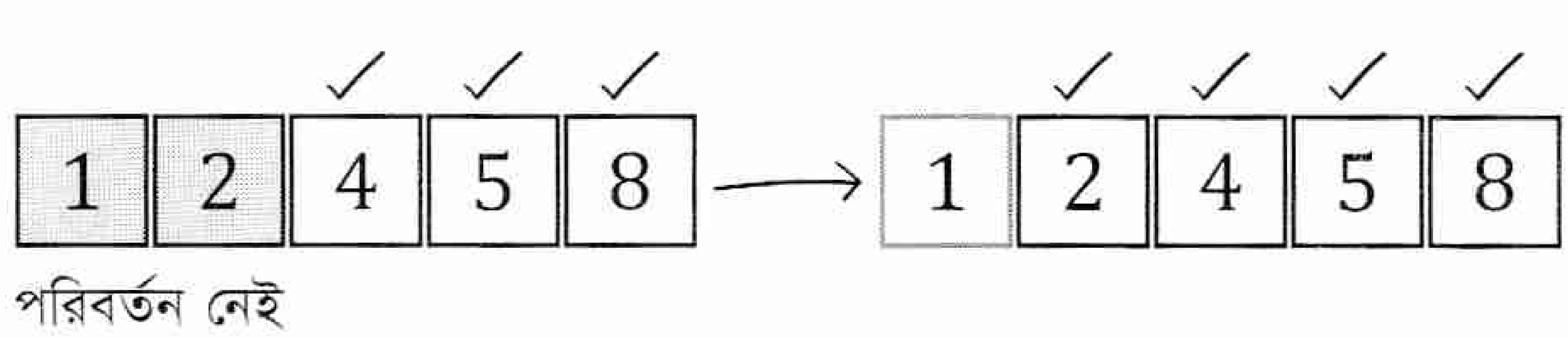
ছবি 4.15

ধাপ 9 : 2 ও 4 তুলনা করি। যেহেতু 2 ছোটো, তাই তাদের স্থানের পরিবর্তন হবে না। তাই সংখ্যাগুলোর অবস্থান আগের মতোই থাকবে– 1, 2, 4, 5, 8 (ছবি 4.16)। এই ধাপের শেষে আমরা বলতে পারি, বড়ো তিনটি সংখ্যা সবচেয়ে শেষ তিনটি স্থানে আছে।



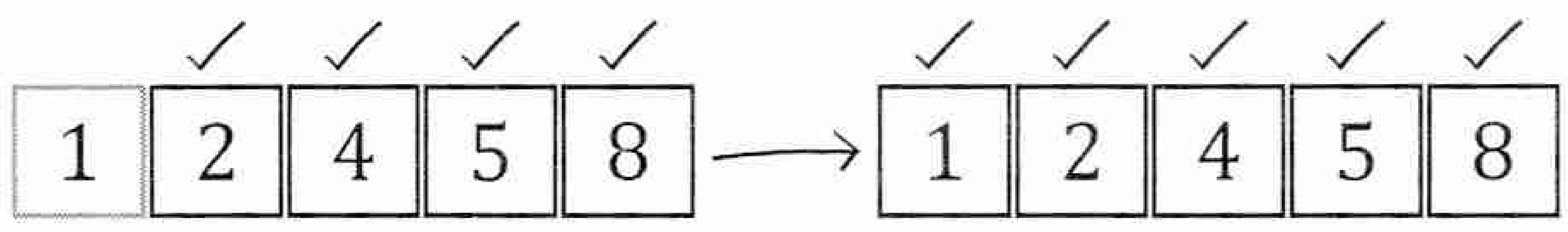
ছবি 4.16

ধাপ 10 : 1 ও 2 তুলনা করি। যেহেতু 1 ছোট, তাই তাদের স্থানের পরিবর্তন হবে না। অর্থাৎ, সংখ্যাগুলোর অবস্থান থাকবে একইরকম– 1, 2, 4, 5, 8 (ছবি 4.17)। এই ধাপের শেষে বলা যায়, বড়ো চারটি সংখ্যা সবচেয়ে শেষ চারটি স্থানে আছে।



ছবি 4.17

ধাপ 11 : যেহেতু আর একটি সংখ্যা অবশিষ্ট আছে (1), তাই এখন আর আমাদের কিছু করতে হবে না (ছবি 4.18)। আমরা এ পর্যায়ে এসে বলতে পারি, সংখ্যাগুলো ছোটো থেকে বড়ো ক্রমে সাজানো হয়ে গিয়েছে।



ছবি 4.18

অনুশীলনী :

পাঁচটি সংখ্যা দেওয়া আছে : 3, 1, 7, 6, 2। সংখ্যাগুলোকে বাবল সর্ট পদ্ধতিতে ছোটো থেকে বড়ো ক্রমে সাজাতে হবে।

ওপরের উদাহরণ ঠিকভাবে বুঝতে পারলে আর অনুশীলনীটি করে ফেললে বাবল সর্ট বুঝতে আর সমস্যা হওয়ার কথা নয়। এখন বাবল সর্ট অ্যালগরিদমটি লিখে ফেলি।

বাবল সর্ট অ্যালগরিদম

ইনপুট : একটি অ্যারে A, যেখানে nসংখ্যক উপাদান আছে এবং উপাদানগুলোর মধ্যে তুলনা করে বড়ো-ছোটো নির্ধারণ করা যায়।

ধাপ 1 : ধরি, `i = 0`।

ধাপ 2 : `i` যদি n-এর চেয়ে ছোটো হয়, তাহলে পরবর্তী ধাপে যাই। না হলে ধাপ 8 (আট)-এ যাই।

ধাপ 3 : ধরি, `j = 0`।

ধাপ 4 : `j` যদি `(n-i-1)`-এর চেয়ে ছোটো হয়, তাহলে পরবর্তী ধাপে যাই। না হলে ধাপ 7-এ যাই।

ধাপ 5 : `A[j]` যদি তার পরবর্তী সংখ্যা (`A[j+1]`)-এর চেয়ে বড়ো হয়, তাহলে তাদের স্থান অদলবদল করি।

ধাপ 6 : `j`-এর মান এক বাড়াই। ধাপ 4-এ যাই।

ধাপ 7 : `i`-এর মান এক বাড়াই। ধাপ 2-এ যাই।

ধাপ 8 : অ্যারে `A`-তে সংখ্যাগুলো ছোটো থেকে বড়ো ক্রমে সাজানো আছে।

তবে এভাবে অ্যালগরিদম লেখা একটু ঝামেলার কাজ। তাই কেউ চাইলে নিচের মতো করে লিখলেও চলবে—

ধাপ 1 : `i`-এর মান `0` থেকে শুরু করে `n`-এর আগ পর্যন্ত এক এক করে বাড়াই এবং প্রতিবার ধাপ 2-এর কাজ করি।

ধাপ 2 : j-এর মান `0` থেকে শুরু করে `(n-i-1)`-এর আগ পর্যন্ত এক এক করে বাড়াই এবং প্রতিবার ধাপ 3-এর কাজ করি।

ধাপ 3 : যদি `A[j] > A[j+1]` হয়, তবে তাদের উপাদানগুলো অদলবদল করি।

ধাপ 4 : অ্যারে `A`-তে সংখ্যাগুলো ছোটো থেকে বড়ো ক্রমে সাজানো আছে।

এখন আমরা অ্যালগরিদমটি ইমপ্লিমেন্ট করব। কারো যদি ওপরের অ্যালগরিদমটি বুঝতে সমস্যা হয়, তাহলে নিচের কোড বোঝার পরে অ্যালগরিদমটি আরেকবার পড়তে হবে। প্রয়োজন হলে চারটি বা পাঁচটি সংখ্যা নিয়ে অ্যালগরিদমটি ঠিকভাবে কাজ করছে কি না, তা খাতা-কলমে করে দেখতে হবে।

```
void bubble_sort(int A[], int n)
{
    int i, j, temp;

    for (i = 0; i < n; i++) {            // ধাপ 1
        for (j = 0; j < n-i-1; j++) {    // ধাপ 2
            if (A[j] > A[j+1]) {             // ধাপ 3
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

অনুশীলনী :

ওপরের প্রোগ্রামে bubble_sort ফাংশনটি এমনভাবে পরিবর্তন করতে হবে, যেন প্রতিটি ধাপের শেষে অ্যারেটি প্রিন্ট হয়।

এখন, বাবল সর্টের টাইম কমপ্লেক্সিটি কত? প্রোগ্রামের দিকে তাকিয়ে আমরা সহজেই বলে দিতে পারি, $O(n \times (n-i-1))$। এখানে `(n-i-1)`-এর মান সবচেয়ে বেশি হবে যখন `i`-এর মান 0 হয়। তখন আমরা লিখতে পারি $O(n \times (n-0-1))$ বা $O(n \times (n-1))$ বা $O(n^2 -$

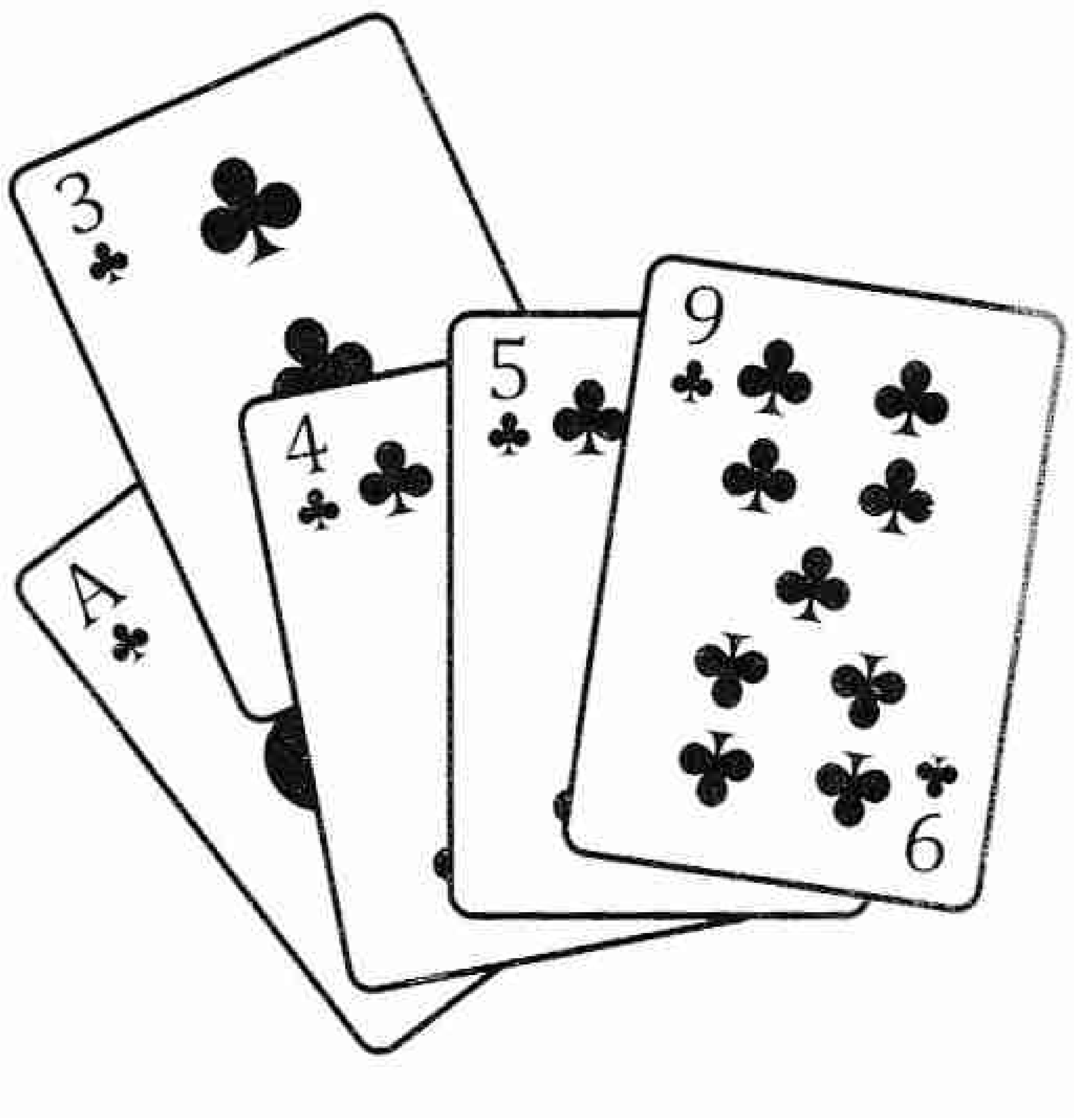
n)। ধারার যোগফলের মাধ্যমেও এটি দেখানো যায়। আর $O(n^2 - n)$-কে আমরা $O(n^2)$ লিখতে পারি। কারণ n^2-এর তুলনায় n বেশ ছোটো, তাই কমপ্লেক্সিটি হিসাব করার সময় সেটিকে উপেক্ষা করা যায়।

অনুশীলনী :

ওপরের প্রোগ্রামটি আবার ভালোভাবে লক্ষ করি। আমরা ভেতরের লুপে একটি তুলনা বা কম্পারিজন অপারেশন করছি (`A[j] > A[j+1]`) এবং সেটির ফলাফল সত্য হলে অদলবদল করার কাজটি করছি। এখন বাইরের লুপের কোনো এক ইটারেশনের সময় যদি আমরা বুঝতে পারি, এই ইটারেশনের সময় ভেতরের লুপের প্রতিটি ইটারেশনে একবারও শর্তটি সত্য হয়নি এবং তাই ভেতরের অদলবদলের কোনো কাজ করা হয়নি, তখন কিন্তু নিশ্চিতভাবে বলা যায় যে, অ্যারেটি ইতিমধ্যেই সাজানো হয়ে গিয়েছে। তাই আর বাইরের লুপটি চালানো দরকার নেই। এর ফলে আমাদের প্রোগ্রামটি আরো ইফিশিয়েন্ট হবে। প্রশ্ন হচ্ছে, কাজটি কীভাবে করতে হবে?

বাবল সর্ট অ্যালগরিদমটি নিয়ে অনেকে অনেক পরীক্ষা-নিরীক্ষা করেছেন এবং বিভিন্নভাবে ইমপ্লিমেন্ট করেছেন। তোমাদের কাজ হবে ইন্টারনেট ঘেঁটে সেগুলো দেখে নেওয়া ও বুঝে নেওয়া।

ইনসার্শন সর্ট (Insertion sort)



ছবি 4.19 : আমরা হাতে কার্ড সাজানোর সময় নিজের অজান্তেই ইনসার্শন সর্ট প্রক্রিয়া ব্যবহার করি

ধরা যাক, একটি অ্যারে কোনো একটি নির্দিষ্ট ক্রমে সাজানো আছে। এখন সেই অ্যারেতে নতুন উপাদান যুক্ত করতে হবে। তবে নতুন উপাদান যোগ করার সময় খেয়াল রাখতে হবে যেন, উপাদানটি যোগ করার পরেও অ্যারেটি সাজানো অবস্থায় থাকে। তাহলে কাজটি কীভাবে করতে

হবে? প্রথমে যেই উপাদানটি আমরা অ্যারেতে যোগ করব, সেটি অ্যারের ঠিক কোন অবস্থানে বসালে সেটি তখনো সাজানো-ই থাকবে, সেই অবস্থানটি খুঁজে বের করতে হবে। যেমন– 1, 4, 5, 9 এই চারটি সংখ্যা ছোটো থেকে বড়ো ক্রমে সাজানো আছে। এখানে 3 সংখ্যাটি যোগ করতে হবে। তাই প্রথমেই খুঁজে বের করতে হবে যে, 3-কে কোথায় বসাতে হবে। যে অবস্থানে বসালে 3-এর আগের অবস্থানের সংখ্যাটি তার চেয়ে ছোটো বা সমান এবং 3-এর পরের অবস্থানের সংখ্যাটি 3-এর চেয়ে বড়ো, সেই অবস্থানটি খুঁজে বের করতে হবে। এ ক্ষেত্রে সে অবস্থানটি হচ্ছে 1-এর পরের অবস্থান। কিন্তু সেখানে তো ইতিমধ্যে 4 আছে। তাই ওই ঘরটি ফাঁকা করতে হবে। এই ফাঁকা করার কাজটি করার জন্য, প্রথমে 9-কে একঘর পরের অবস্থানে সরাতে হবে– 1, 4, 5, □, 9। তাহলে 9-এর আগের অবস্থানে একটি ফাঁকা জায়গা তৈরি হবে। তারপরে 5-কে তার পরের অবস্থানের ফাঁকা ঘরে রাখতে হবে : 1, 4, □, 5, 9। তারপরে 4-কে একঘর পরের অবস্থানে সরাতে হবে : 1, □, 4, 5, 9। এখন 3-কে যেখানে বসাতে চাই, সেই জায়গাটি ফাঁকা হয়ে গেল। তাই 3-কে ওখানে বসাতে হবে– 1, 3, 4, 5, 9। এই পদ্ধতিতে সাজানোকেই বলে ইনসার্শন সর্ট (Insertion Sort)।

এখন পুরো ইনসার্শন সর্ট প্রক্রিয়ার উদাহরণ দেওয়া যাক। 5, 2, 3, 1, 4 – এই পাঁচটি সংখ্যাকে ইনসার্শন সর্ট প্রক্রিয়া ব্যবহার করে সাজাতে হবে (ছবি 4.20)।

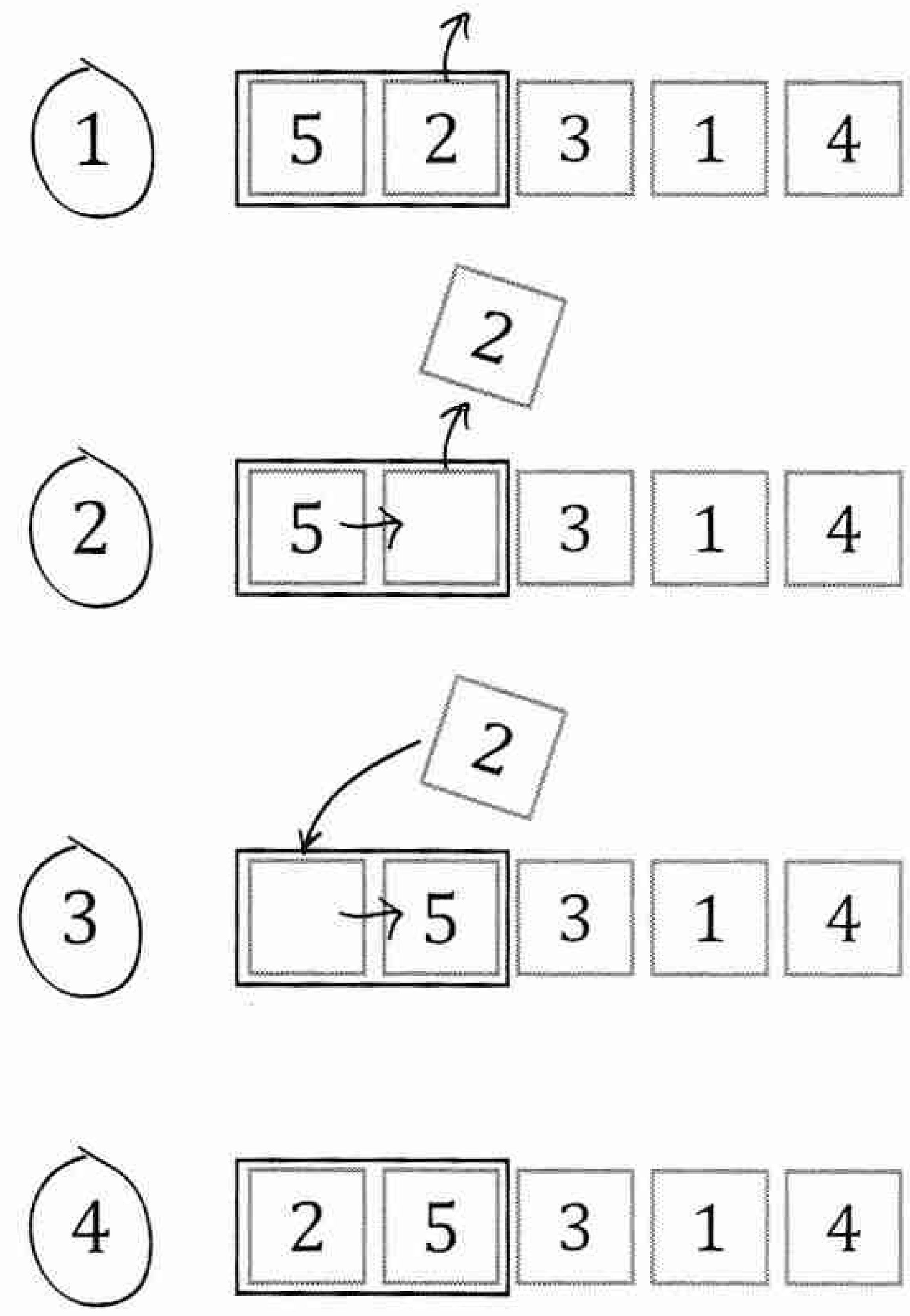
5	2	3	1	4

ছবি 4.20

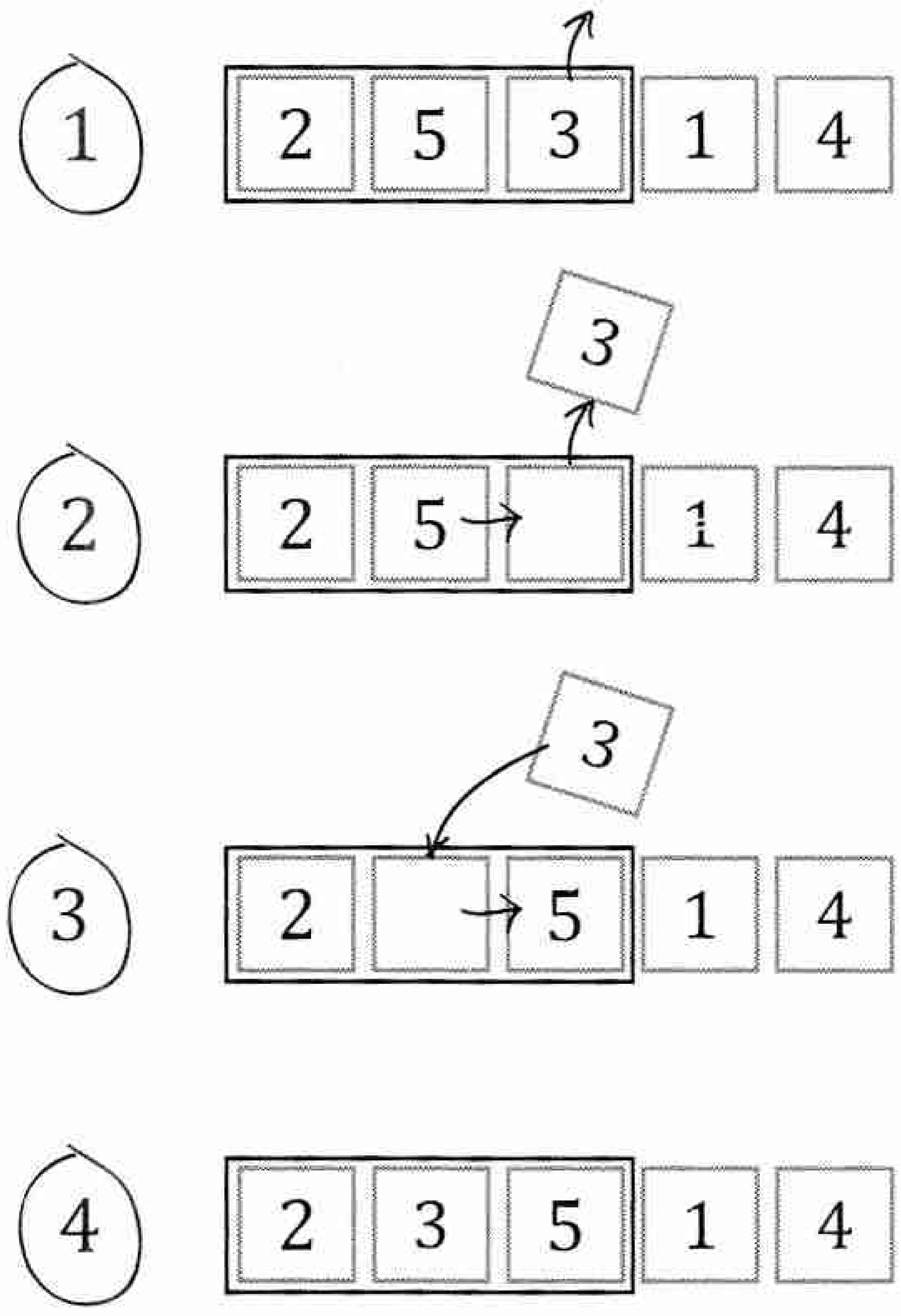
আমরা যেটি করব, প্রথমে অ্যারের প্রথম দুটি উপাদান সর্ট করে ফেলব ইনসার্শন সর্ট পদ্ধতির মাধ্যমে। তারপরে সেখানে তৃতীয় উপাদানটি যুক্ত করব এবং ইনসার্শন সর্ট করে সাজিয়ে ফেলব। তাহলে অ্যারের প্রথম তিনটি উপাদান সাজানো থাকবে। তারপরে চতুর্থ উপাদানটি যোগ করব এবং একইভাবে সাজিয়ে ফেলব। তাহলে প্রথম চারটি উপাদান সাজানো থাকল। পঞ্চম উপাদানের ক্ষেত্রেও একই কাজ করব।

ধাপ 1 :**5, 2**, 3, 1, 4; এখানে প্রথমে 2-এর জন্য উপযুক্ত জায়গা খুঁজে বের করি, সেটি হচ্ছে প্রথম স্থান আর 5 থাকবে দ্বিতীয় স্থানে। তাহলে 5-কে একঘর ডান দিকে সরাই– □, 5, 3, 1, 4। তারপর 2-কে তার সঠিক স্থানে বসাই– 2, 5, 3, 1, 4 (ছবি 4.21)।

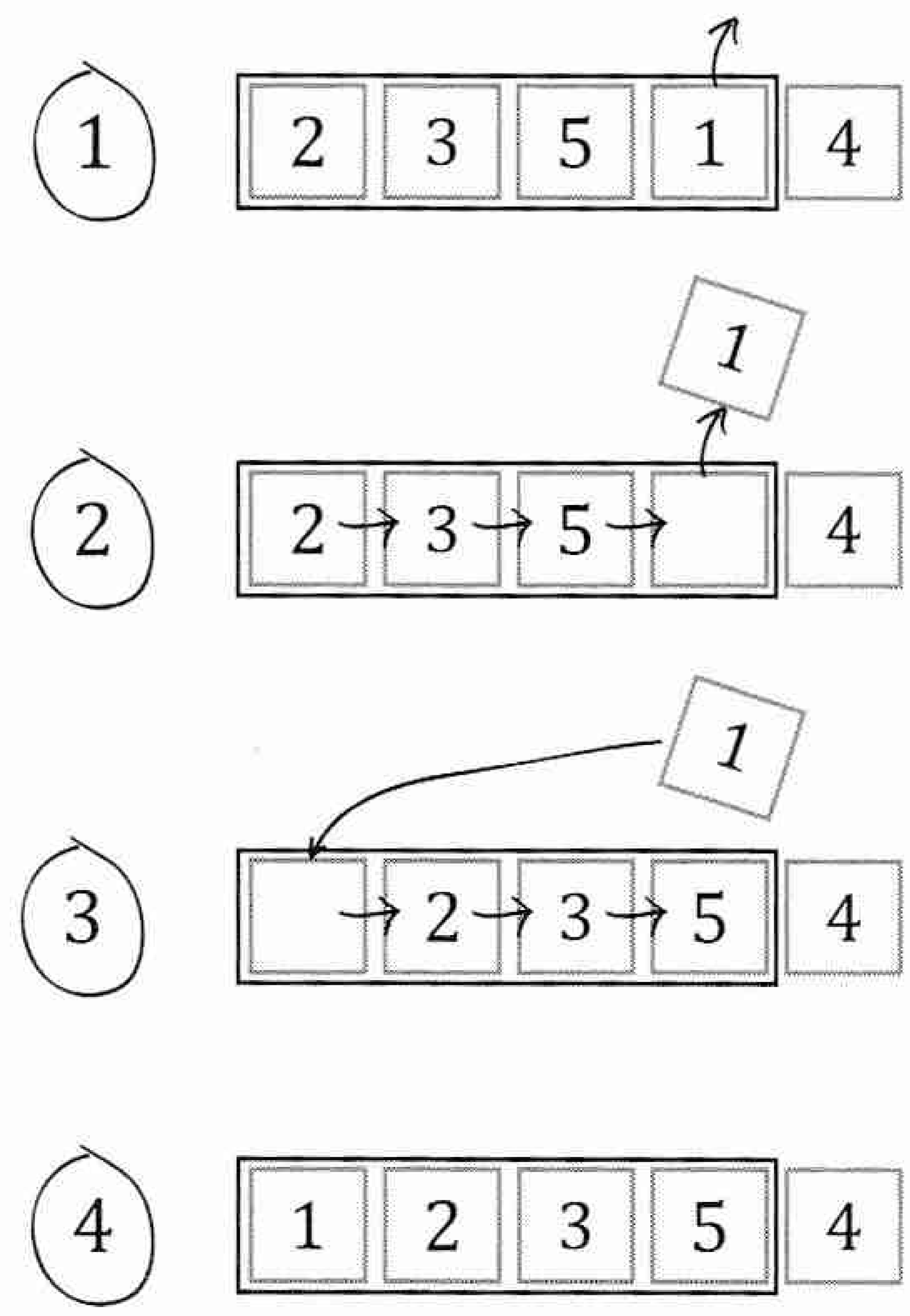
ধাপ 2 : **2, 5**, 3, 1, 4; এখানে 3-এর জন্য উপযুক্ত জায়গা খুঁজে বের করি, সেটি হচ্ছে 2 ও 5-এর মাঝখানে। তাই 5-কে এক ঘর ডান দিকে সরাই : 2, □, 5, 1, 4। তারপর 3-কে ফাঁকা স্থানে বসিয়ে দিই– 2, 3, 5, 1, 4 (ছবি 4.22)।



ছবি 4.21



ছবি 4.22



ছবি 4.23

ধাপ 3 : **2, 3, 5**, 1, 4; এবারে 1-এর জন্য উপযুক্ত স্থান খুঁজে বের করতে হবে। 1-এর উপযুক্ত স্থান হচ্ছে সবার প্রথমে, অর্থাৎ 2-এর স্থানে। তাই 2, 3, 5 প্রত্যেককে ডান দিকে এক ঘর সরাই—

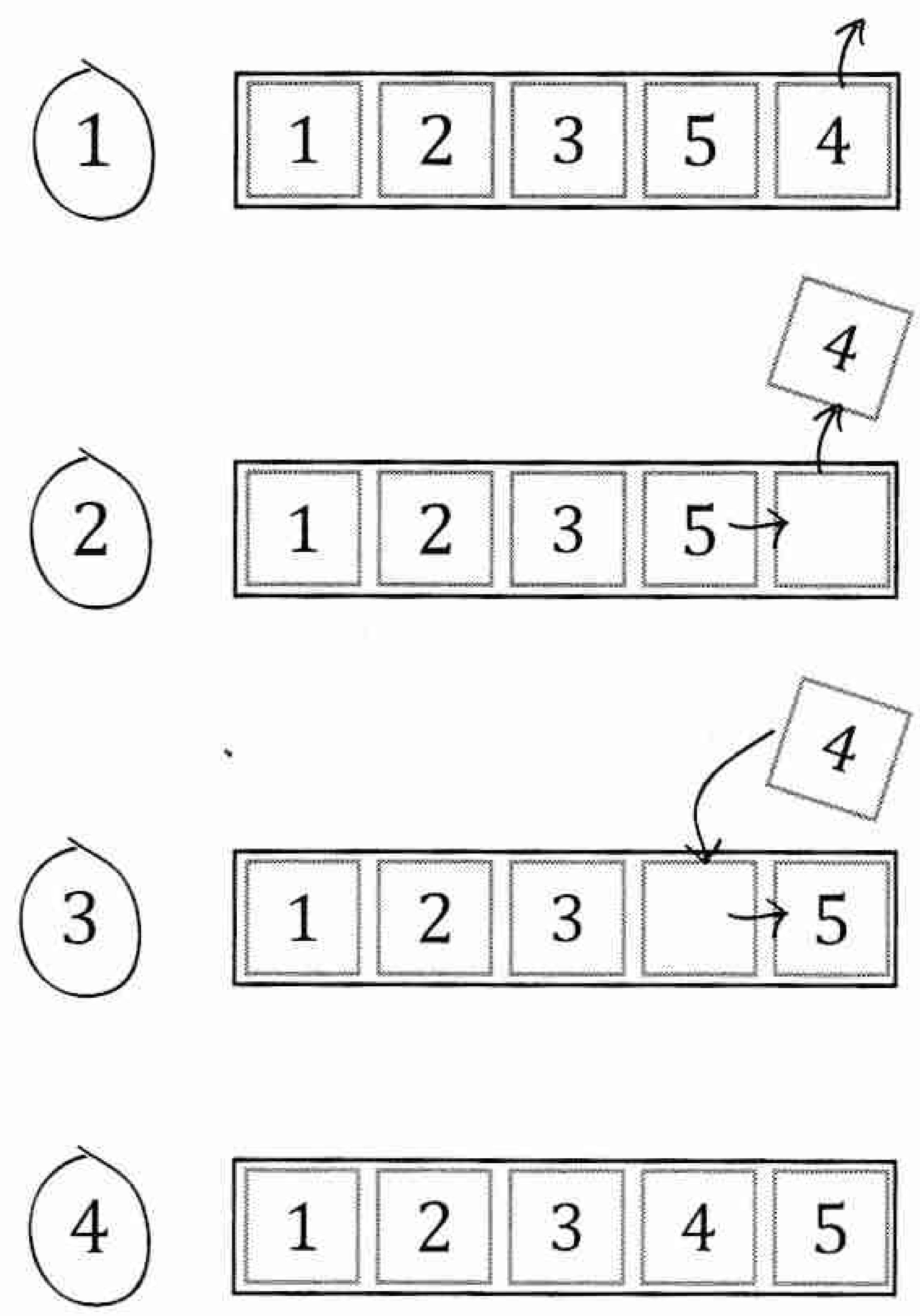
2, 3, □, 5, 4

2, □, 3, 5, 4

☐, 2, 3, 5, 4

তাহলে প্রথম ঘরটি ফাঁকা হলো, সেখানে 1 বসাই– 1, 2, 3, 5, 4 (ছবি 4.23)।

ধাপ 4 : **1, 2, 3, 5**, 4 - শেষ সংখ্যা 4-কে তার সঠিক স্থানে বসাতে হবে। 4-এর সঠিক স্থান হচ্ছে 3-এর পরে ও 5-এর আগে। তাই 5-কে এক ঘর ডান দিকে সরাই : 1, 2, 3, ☐, 5। এবারে ফাঁকা জায়গায় 4-কে বসিয়ে দিই : 1, 2, 3, 4, 5 (ছবি 4.24)। সংখ্যাগুলো ছোটো থেকে বড়ো ক্রমে সাজানো হয়ে গেল।



ছবি 4.24

অনুশীলনী :

1. 5, 4, 3, 2, 1 – সংখ্যাগুলোকে ইনসার্শন সর্ট ব্যবহার করে ছোটো থেকে বড়ো ক্রমে সাজাতে হবে।
2. 1, 2, 3, 4, 5 – সংখ্যাগুলোকে ইনসার্শন সর্ট ব্যবহার করে ছোটো থেকে বড়ো ক্রমে সাজাতে হবে।

ওপরের দুটি অনুশীলনীর প্রথমটি করার সময় একটু বেশি কষ্ট করতে হবে। দেখা যাবে, খাতায় বেশ জায়গা ব্যবহার করার প্রয়োজন হচ্ছে। এর কারণ হচ্ছে সংখ্যাগুলো একেবারে উলটোভাবে সাজানো। তাই প্রতিবারই একটি সংখ্যাকে তার সঠিক স্থানে বসানোর জন্য আগের সব সংখ্যাগুলোকে এক ঘর করে ডান দিকে সরতে হবে। আবার দ্বিতীয় অনুশীলনী করার সময় দেখা

যাবে খুব অল্পতেই কাজ হয়ে যাচ্ছে। কারণ সংখ্যাগুলো সঠিক ক্রমে সাজানো আছে। তাই কোনো সংখ্যাকেই তার জন্য নির্ধারিত স্থানে রাখার প্রয়োজনে আগের সংখ্যাগুলোকে সরাতে হবে না। তাই যখন কোনো অ্যারে ইতিমধ্যেই সাজানো আছে বা অ্যারের বেশিরভাগ উপাদানই সঠিক ক্রমে সাজানো আছে, তখন ইনসার্শন সর্ট ব্যবহার করলে প্রোগ্রাম খুব দ্রুত কাজ করবে। তবে কমপ্লেক্সিটি হিসাব করার সময় আমরা সবচেয়ে প্রতিকূল পরিস্থিতি, অর্থাৎ ওয়ার্স্ট কেস বেছে নেব।

এখন ইনসার্শন সর্ট করার জন্য একটি ফাংশন লিখে ফেলি। তারপরে আমরা অ্যালগরিদম লিখব।

```
void insertion_sort(int A[], int n)
{
    int i, j, item;

    for (i = 1; i < n; i++) {
        // A[i]-কে item-এ অ্যাসাইন করি
        item = A[i];

        // এখন item-এর জন্য উপযুক্ত স্থান খুঁজে বের করতে হবে
        j = i - 1;

        while (j >= 0 && A[j] > item) {
            // A[j]-কে তার পরের ঘরে (j+1) রেখে দিই
            A[j+1] = A[j];
            j = j - 1;
        }

        // A[j+1] হচ্ছে item-এর জন্য উপযুক্ত স্থান
        // এবং সেটি ইতিমধ্যে আমরা খালি করে রেখেছি
        A[j+1] = item;
    }
}
```

ওপরের প্রোগ্রামে, বাইরের লুপের ভেতরে প্রথমে `item = A[i]` লিখেছি। এতে আমি নিশ্চিত করছি যে, `A[i]`-তে যদি অন্য কিছু রাখি, তাহলে `A[i]`-এর আগের মানটি হারিয়ে যাবে না, বরং `item`-এর মধ্যে থাকবে। এ ছাড়া এই `item`-কেই তো আমাদের সঠিক স্থানে রাখতে হবে। ভেতরের লুপটি লিখেছি এভাবে— `while (j >= 0 && A[j] > item)`। প্রথমে `j >= 0` লেখার কারণ হচ্ছে, যেহেতু আমি `j`-এর মান সেই লুপের ভেতরে এক করে কমাচ্ছি (`j = j - 1`), তাই সেটি কমতে কমতে যদি -1 হয়ে যায়, তখন তো আমি বুঝতে পারব যে,

লিস্টের সবাই-ই item-এর চেয়ে বড়ো। আর অ্যারের ইনডেক্সে -1 ব্যবহার করলে কম্পাইলার এরর দিবে। তাই প্রথমে j >= 0 লিখলাম, যেন শর্তটি মিথ্য হলে লুপ থেকে বের হয়ে যায় এবং A[j] > item এই পরীক্ষার কাজটি না হয়। আর শর্তটি সত্য হলে, দ্বিতীয় শর্ত, অর্থাৎ A[j] > item পরীক্ষা করা হয়। এখন A[j] যদি item-এর চেয়ে বড়ো হয়, তাহলে item অবশ্যই A[j]-এর আগের কোনো ঘরে বসবে। তাই A[j]-কে আমরা এক ঘর ডানে সরিয়ে দিই, মানে A[j+1]-এ A[j]-কে রাখি (A[j+1] = A[j])। আর A[j] > item শর্তটি যদি মিথ্যা হয়, তাহলে আমরা জানি যে, item-এর স্থান হচ্ছে j+1-তম ঘরে, কারণ তার আগের সবগুলো সংখ্যা তার চেয়ে ছোটো কিংবা সমান। তাই লুপ থেকে বের হয়ে আমরা A[j+1]-এ item-কে রেখে দিচ্ছি। আবার যদি j >= 0 শর্তটি মিথ্যা হয়, অর্থাৎ j-এর মান কমতে কমতে -1 হয়, তখন আমরা জানি যে, অ্যারের সব উপাদানই item-এর চেয়ে বড়ো, তাই অ্যারের শুরুতে (অর্থাৎ 0-তম ঘরে) item-কে রাখতে হবে। তাই, A[j + 1] বা A[-1 + 1] বা A[0] ঘরে item-কে রেখে দিই।

এখন আমরা ইনসার্শন সর্টের অ্যালগরিদমটি লিখব।

ইনসার্শন সর্ট অ্যালগরিদম

ইনপুট : একটি অ্যারে A, যেখানে nসংখ্যক উপাদান আছে এবং উপাদানগুলোর মধ্যে তুলনা করে বড়ো-ছোটো নির্ধারণ করা যায়।

ধাপ 1 : অ্যারেতে একটি উপাদান থাকলে সেটি ইতিমধ্যে সর্ট করা। তাই অ্যারের দ্বিতীয় উপাদান থেকে শুরু করে শেষ পর্যন্ত প্রতিটি উপাদানের জন্য পরবর্তী দুটি ধাপের কাজ করি।

ধাপ 2 : একটি নির্দিষ্ট উপাদানের জন্য তার আগে অ্যারেতে যে সব উপাদান আছে, সেগুলো যদি তার চেয়ে বড়ো হয়, তাহলে তাদেরকে এক ঘর করে ডানে সরাই। তাহলে আমরা ওই নির্দিষ্ট উপাদানের জন্য কাঙ্ক্ষিত জায়গা পেয়ে যাব।

ধাপ 3 : নির্দিষ্ট উপাদানটি তার কাঙ্ক্ষিত জায়গায় বসাই।

ধাপ 4 : অ্যারে A ছোটো থেকে বড়ো ক্রমে সাজানো আছে।

বাবল সর্ট ও সিলেকশন সর্টের মতো, ইনসার্শন সর্টের কমপ্লেক্সিটিও হচ্ছে $O(n^2)$। তবে, লিস্ট যদি আগে থেকেই সর্ট কথা থাকে, তাহলে ইনসার্শন সর্টের কমপ্লেক্সিটি হবে $O(n)$, মানে এটি হচ্ছে ইনসার্শন সর্টের বেস্ট কেইস কমপ্লেক্সিটি। এই বিষয়টি আমাদের মাথায় রাখতে হবে।

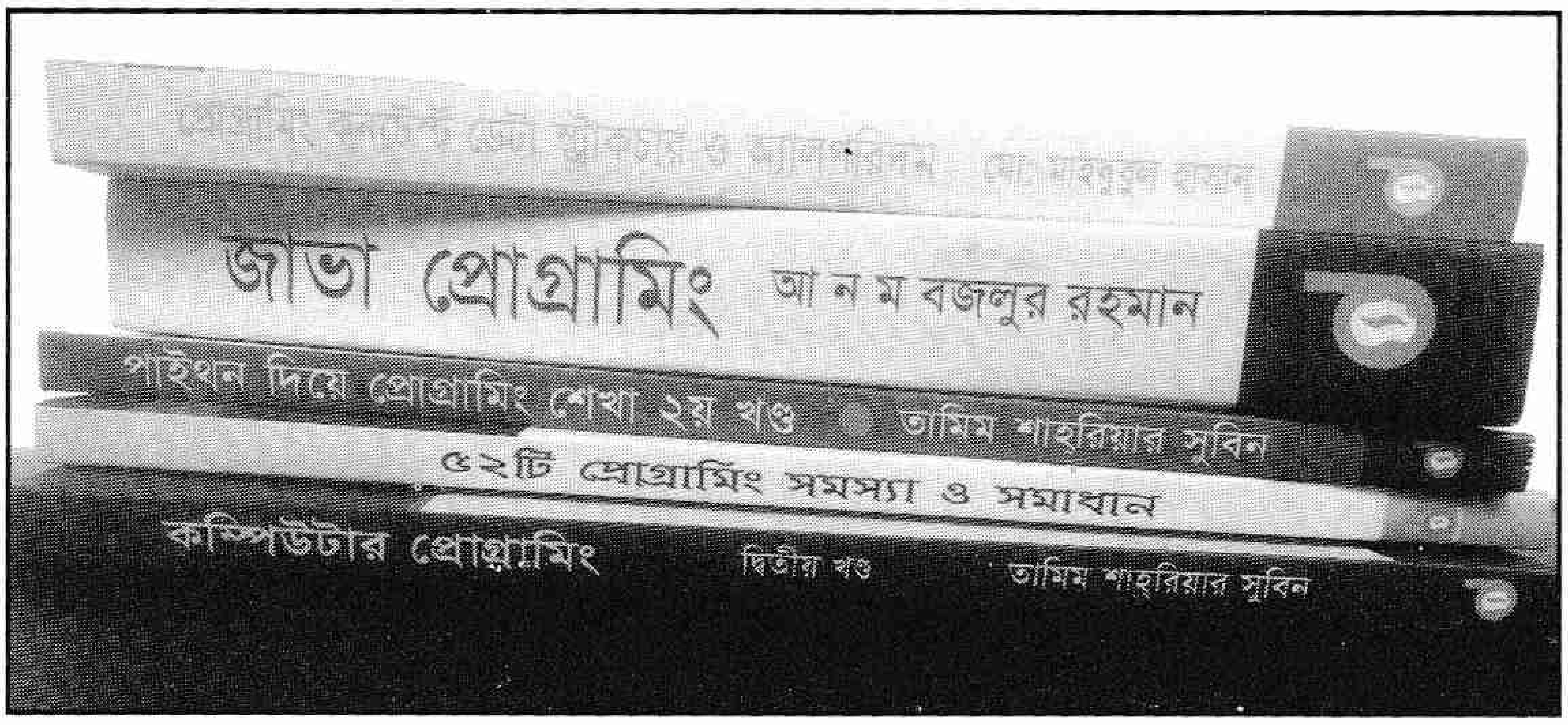
অধ্যায় ৫ : স্ট্যাক (Stack) ও কিউ (Queue)

এই অধ্যায়ে আমরা দুটি ডেটা স্ট্রাকচার শিখব, সেগুলো হচ্ছে স্ট্যাক ও কিউ। এই ডেটা স্ট্রাকচারগুলো শেখার সময় দুটি জিনিস বিশেষভাবে খেয়াল রাখতে হবে–

1) ডেটা স্ট্রাকচারে কীভাবে নতুন ডেটা যোগ করতে হয়?
2) ডেটা স্ট্রাকচার থেকে কীভাবে ডেটা নিতে হয়?

স্ট্যাক (Stack)

স্ট্যাক (stack) শব্দের বাংলা অর্থ হচ্ছে স্তূপ। স্তূপের বৈশিষ্ট্য কী? একটি স্তূপে আমরা যখন কোনো জিনিস রাখি, তখন সেটি সবার ওপরে রেখে দিই (কেউ কেউ অবশ্য একটু অন্যভাবেও রাখার চেষ্টা করে, তবে সেটি ব্যতিক্রম)। আবার যখন স্তূপ থেকে কোনো কিছু সরাই, তখন সবচেয়ে ওপরে যেই জিনিসটি আছে, সেটি সরাই (ছবি 5.1)।



ছবি 5.1

স্ট্যাক ডেটা স্ট্রাকচারেও ডেটাগুলো এমনভাবে রাখা হয়, যেন, কেবল সবার শেষের ডেটা সরানো যায়, আর নতুন ডেটা যোগ করতে হলে সেটি সবার শেষে এসে যুক্ত হয়।

স্ট্যাক তৈরি করার জন্য যদি একটি অ্যারে ব্যবহার করি, তাহলে স্ট্যাকে চারটি সংখ্যা রাখলে স্ট্যাকটি দাঁড়াবে ছবি 5.2-এর মতো।

এখন আমি যদি এই স্ট্যাকে আরেকটি সংখ্যা, যেমন 6 রাখতে চাই, তাহলে সেটি রাখতে হবে অ্যারের একেবারে শেষে। ছবি 5.3-এর মতো।

4	5	2	3			

ছবি 5.2

4	5	2	3	6		

ছবি 5.3

এখন আমরা স্ট্যাকে যে অ্যারে ব্যবহার করব, সেই অ্যারের কোথায় নতুন ডেটা যোগ করতে হবে, তার হদিস রাখার জন্য আমাদের একটি ভ্যারিয়েবল ব্যবহার করতে হবে। তার নাম আমরা দিতে পারি top। স্তূপের সবচেয়ে ওপরে যেমন নতুন জিনিস রাখা হয়, তেমনি আমরা স্ট্যাকের জন্য যেই অ্যারে ব্যবহার করব, তার সবচেয়ে ওপরে বা top ইনডেক্সে ডেটা রাখব। তাহলে সি দিয়ে একটি স্ট্রাকচার তৈরি করে ফেলা যাক।

```
#define STACK_MAX 100

typedef struct {
    int top;
    int data[STACK_MAX];
} Stack;
```

এখানে STACK_MAX আমরা স্ট্যাকে সর্বোচ্চ কয়টি ডেটা রাখব সেটি নির্দেশ করে। আর আমরা typedef ব্যবহার করে স্ট্রাকচারটির নাম দিচ্ছি Stack, যাতে আমরা স্ট্যাক তৈরি করতে হলে Stack s1 – এভাবে তৈরি করতে পারি। আর স্ট্যাক তৈরি করার পরে শুরুতেই তার top এর মান 0 করে দিতে হবে। top-এর মান 0-এর মানে হচ্ছে স্ট্যাকে কোনো ডেটা নেই।

এখন আমরা স্ট্যাকে ডেটা রাখা আর স্ট্যাক থেকে ডেটা নেওয়ার পদ্ধতি আলোচনা করব। কম্পিউটার বিজ্ঞানের ভাষায়, স্ট্যাক থেকে ডেটা নেওয়াকে বলে pop (পপ), আর স্ট্যাকে ডেটা রাখাকে বলে push (পুশ)।

স্ট্যাকে ডেটা রাখার সময় আমরা প্রথমে পরীক্ষা করব যে, স্ট্যাকে জায়গা আছে কি না। যদি জায়গা থাকে, তাহলে স্ট্যাকের top ইনডেক্সে ডেটা রাখব। তারপরে top-এর মান এক বাড়িয়ে দেব। আর স্ট্যাকে যদি জায়গা না থাকে, তাহলে বলে দিব যে, স্ট্যাকে কোনো জায়গা নেই।

স্ট্যাক থেকে ডেটা পড়ার সময় কী করতে হবে? আমরা জানি, top ইনডেক্সের আগের ইনডেক্স পর্যন্ত ডেটা আছে। তাই top-এর মান এক কমিয়ে দেব, এবং তারপরে top ইনডেক্সে যা আছে, সেটি রিটার্ন করে দেব। এখন তোমরা প্রশ্ন করতে পারো, রিটার্ন করলে তো আর স্ট্যাক থেকে ডেটা মুছে যাচ্ছে না, সেই ডেটার কী হবে? আমরা যখন top-এর মান কমিয়ে দিচ্ছি, তার মানে হচ্ছে আমরা যদি এরপর স্ট্যাকে কোনো ডেটা রাখতে চাই, তাকে top ইনডেক্সের ঘরে রাখা হবে। তাই আগে কি ডেটা ছিল, তা নিয়ে আমাদের চিন্তা করতে হবে না, কারণ ডেটা ওভাররাইট হয়ে যাবে। আরেকটি কথা বলে নিই, স্ট্যাক থেকে ডেটা পড়ার আগে আমাদের একটি জিনিস

পরীক্ষা করা উচিত। তোমরা কি বলতে পারবে সেই জিনিসটি কী? কয়েক মিনিট চিন্তা করে দেখো বলতে পারো কি না।

এখন আমরা স্ট্যাকের কাজগুলোর জন্য আলাদা ফাংশন তৈরি করে ফেলি। তারপরে মেইন ফাংশন থেকে স্ট্যাক তৈরি করে সেই ফাংশনগুলো ব্যবহার করে দেখব। আমি পুরো প্রোগ্রামটি লিখে দিচ্ছি। আমি ফাংশনের প্যারামিটার হিসেবে পয়েন্টার ব্যবহার করছি। তোমাদের যদি স্ট্রাকচার, পয়েন্টার এসব জিনিস ভালোভাবে জানা না থাকে, তাহলে তোমরা আমার লেখা কম্পিউটার প্রোগ্রামিং ২য় খণ্ড বইটি পড়ে নিতে পারো। অন্য কোনো বই থেকে পড়লেও চলবে।

```
#include <stdio.h>

#define STACK_MAX 10

typedef struct {
    int top;
    int data[STACK_MAX];
} Stack;

// আমরা প্যারামিটার হিসেবে একটি স্ট্যাকের পয়েন্টার (*s) ও একটি ইন্টিজার
// (item) ব্যবহার করছি। আমাদের লক্ষ্য হচ্ছে, s-এ item-কে রাখা। ফাংশনটি
// কোনো কিছু রিটার্ন করবে না।
void push(Stack *s, int item)
{
    // প্রথমে পরীক্ষা করে নিচ্ছি স্ট্যাকে জায়গা আছে কি না।
    if (s->top < STACK_MAX) {
        // স্ট্যাকের সবচেয়ে ওপরে অর্থাৎ s->top ইনডেক্সে item রেখে দিচ্ছি।
        s->data[s->top] = item;
        // s-top-এর মান এক বাড়িয়ে দিই।
        s->top = s->top + 1;
    } else {
        // স্ট্যাকে জায়গা না থাকলে নিচের লাইনটি প্রিন্ট করি।
        printf("Stack is full!\n");
    }
}

// এখানে প্যারামিটার হিসেবে একটি স্ট্যাকের পয়েন্টার ব্যবহার করছি।
// ফাংশনটি সেই স্ট্যাকের সবচেয়ে ওপরের উপাদান বা ডেটা রিটার্ন করবে।
int pop(Stack *s)
```

```
{
    int item;

    // প্রথমে পরীক্ষা করে নিচ্ছি যে, স্ট্যাকটি খালি কি না।
    if (s->top == 0) {
        // স্ট্যাক খালি, তাই একটি মেসেজ প্রিন্ট করে -1 রিটার্ন করে ফাংশন
        // থেকে বের হয়ে যাব।
        printf("Stack is empty!\n");
        return -1;
    } else {
        // s->top-এর মান এক কমিয়ে দিচ্ছি।
        s->top = s->top - 1;
        // এখন s->top ইনডেক্সে যেই ডেটা আছে, সেটিই আমাদের কাঙ্ক্ষিত
        // উপাদান।
        item = s->data[s->top];
    }

    return item;
}

int main()
{
    // my_stack নামে একটি স্ট্যাক তৈরি করি।
    Stack my_stack;
    int item;

    // শুরুতে my_stack-এ কিছু নেই, তাই my_stack.top-এর মান 0
    // অ্যাসাইন করি।
    my_stack.top = 0;

    // my_stack-এ যথাক্রমে 1, 2 ও 3 রাখি। লক্ষ্য করো যে, প্রথম আর্গুমেন্ট
    // হিসেবে my_stack-এর অ্যাড্রেস পাঠাচ্ছি : &my_stack, অর্থাৎ একটি
    // Stack টাইপের পয়েন্টার পাঠাচ্ছি।
    push(&my_stack, 1);
    push(&my_stack, 2);
    push(&my_stack, 3);

    // এবারে পরপর তিনবার pop() ফাংশন কল করব এবং এটি যে মান রিটার্ন
    // করবে সেটি প্রিন্ট করব।
```

```
    // pop() ফাংশনটি প্যারামিটার হিসেবে স্ট্যাকের পয়েন্টার নেয়,  তাই আমরা
    // &my_stack পাঠাচ্ছি।
    item = pop(&my_stack);
    printf("%d\n", item);

    item = pop(&my_stack);
    printf("%d\n", item);

    item = pop(&my_stack);
    printf("%d\n", item);
}
```

তোমাদের কাজ হচ্ছে প্রোগ্রামটি টাইপ করে কম্পাইল ও রান করা। প্রোগ্রামটি রান করলে তোমরা দেখবে, স্ট্যাকে যেই ক্রমে ডেটা রাখা হয়েছে, তার বিপরীত ক্রমে স্ট্যাক থেকে ডেটা পাওয়া যাচ্ছে। এখন তোমরা কয়েকটি প্রশ্নের উত্তর দাও–

1) pop() ও push() ফাংশনে প্যারামিটার হিসেবে Stack টাইপের ভ্যারিয়েবল (Stack s) ব্যবহার না করে পয়েন্টার (Stack *s) কেন ব্যবহার করা হলো? ভ্যারিয়েবল ব্যবহার করলে কী সমস্যা হতো?
2) আমরা যদি মেইন ফাংশনে আরো একবার pop() ফাংশন কল করতাম, তাহলে আমরা কী পেতাম?
3) আমরা যেই স্ট্যাক তৈরি করেছি, তাতে সর্বোচ্চ কয়টি সংখ্যা রাখা যাবে?

এখন প্রশ্ন হচ্ছে, স্ট্যাকে যে আমরা পুশ ও পপ অপারেশন করি, এদের কমপ্লেক্সিটি কত? উভয় ফাংশনের কমপ্লেক্সিটি হচ্ছে $O(1)$। কারণ স্ট্যাকে যতগুলো উপাদানই থাকুক না কেন, আমরা কেবল শেষে একটি উপাদান যোগ করছি। আর pop() মেথড ব্যবহার করে যখন স্ট্যাকের একেবারে শেষ উপাদানটি আমরা নিয়ে নিচ্ছি– সেখানে যতগুলো উপাদানই থাকুক না কেন, বাকিদের কোনো পরিবর্তন হচ্ছে না।

স্ট্যাকের খুব সাধারণ একটি ব্যবহার হচ্ছে, অনেকগুলো ব্র্যাকেট চিহ্নবিশিষ্ট একটি স্ট্রিং দেওয়া আছে, সেগুলো ঠিকঠাকভাবে জোড়ায় জোড়ায় আছে কি না সেটি পরীক্ষা করা। যেমন : (()- এখানে ব্র্যাকেটগুলো ঠিক নেই। আবার ((()))- এখানে ব্র্যাকেটগুলো ঠিকভাবে আছে। আমরা এখন কাজটি করার জন্য একটি প্রোগ্রাম লিখব– এখানে আমি স্ট্যাকের জন্য আলাদা টাইপ তৈরি না করে কেবল একটি অ্যারে ব্যবহার করব। আর সেই সঙ্গে top নামে একটি ভ্যারিয়েবল প্রয়োজন হবে।

```
int is_balanced(char input[])
```

```
{
    char stack[100], last_char;
    int top, i, len;

    top = 0;
    len = strlen(input);

    for (i = 0; i < len; i++) {
        if (input[i] == '(') {
            stack[top] = '(';
            top++;
        } else if (input[i] == ')') {

            if (top == 0) {
                // stack empty
                return 0;
            }

            top--;

            last_char = stack[top];
            if (last_char != '(') {
                return 0;
            }
        }
    }

    if (top == 0) {
        return 1;
    } else {
        return 0;
    }
}

int main()
{
    char input[100];

    scanf("%s", input);

    if (is_balanced(input)) {
```

```
        printf("%s is balanced\n", input);
    } else {
        printf("%s is not balanced\n", input);
    }

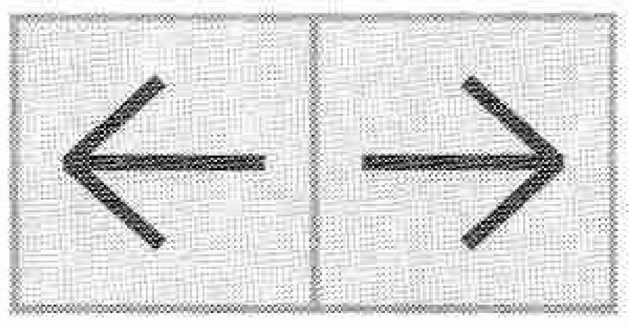
    return 0;
}
```

তোমরা ওপরের প্রোগ্রামটি রান করে বিভিন্ন ইনপুট দিয়ে দেখো কী আউটপুট আসে। আশা করি, প্রোগ্রামটি বুঝতে কোনো সমস্যা হবে না।

এখন কয়েকটি ইংরেজি শব্দের সঙ্গে পরিচয় করিয়ে দিই। ( ) বা প্রথম বন্ধনীকে ইংরেজিতে বলে parentheses (প্যারেন্থেসেস), { } বা দ্বিতীয় বন্ধনীকে ইংরেজিতে বলে curly braces (কার্লি ব্রেইসেস) আর [ ] বা তৃতীয় বন্ধনীকে ইংরেজিতে বলে square brackets (স্কয়ার ব্র্যাকেটস)।

অনুশীলনী :

1. ওপরের প্রোগ্রামটি এমনভাবে লিখতে হবে, যেন, সেটি প্রথম, দ্বিতীয় ও তৃতীয় বন্ধনীর জন্য কাজ করে। যেমন, ইনপুট যদি {{([])}} দেওয়া হয়, তাহলে প্রোগ্রামটি বলবে যে এটি ঠিক আছে। আবার ইনপুট যদি]{{{[)}} হয়, তাহলে প্রোগ্রামটি বলবে যে এটি ঠিক নেই।
2. ওয়েব ব্রাউজারের ব্যাক ও ফরোয়ার্ড বাটনের সঙ্গে নিশ্চয় সবাই পরিচিত (ছবি 5.4)। আমরা যখন ব্রাউজারে বিভিন্ন ওয়েবসাইট ব্যবহার করি, তখন আগের ওয়েবসাইটে বা ওয়েবপেজে যেতে চাইল ব্যাক বাটন ব্যবহার করতে হয়। আবার আগের ওয়েবসাইটে বা ওয়েবপেজে গেলে, আবার পরের ওয়েবপেজে ফিরে আসার জন্য ফরোয়ার্ড বাটন ব্যবহার করা হয়। এই কাজটি কিন্তু দুটি স্ট্যাক ব্যবহার করে সহজেই করা যায়। কীভাবে করতে হবে সেটি ব্যাখ্যা করতে হবে।

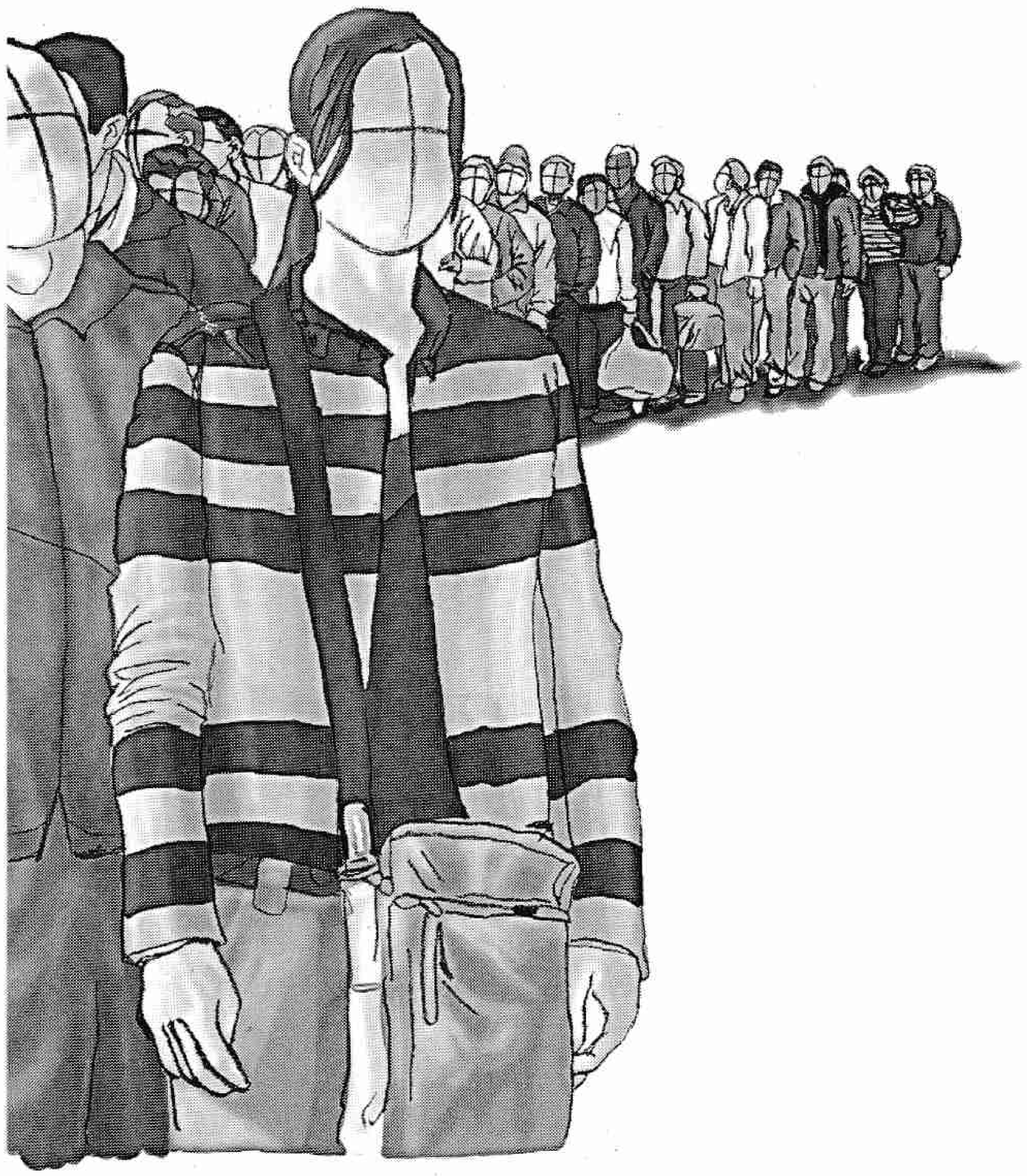


ছবি 5.4

আবার বিভিন্ন সফটওয়্যারে Undo ও Redo বাটন থাকে। যারা এর সঙ্গে পরিচিত, তারা এখন বুঝতে পারবে যে, এই কাজটিও স্ট্যাক ব্যবহার করে করা সম্ভব। স্ট্যাকে যে সবার শেষে আসে, সে সবার আগে চলে যায়, তাই একে Last In First Out (সংক্ষেপে LIFO) বলে।

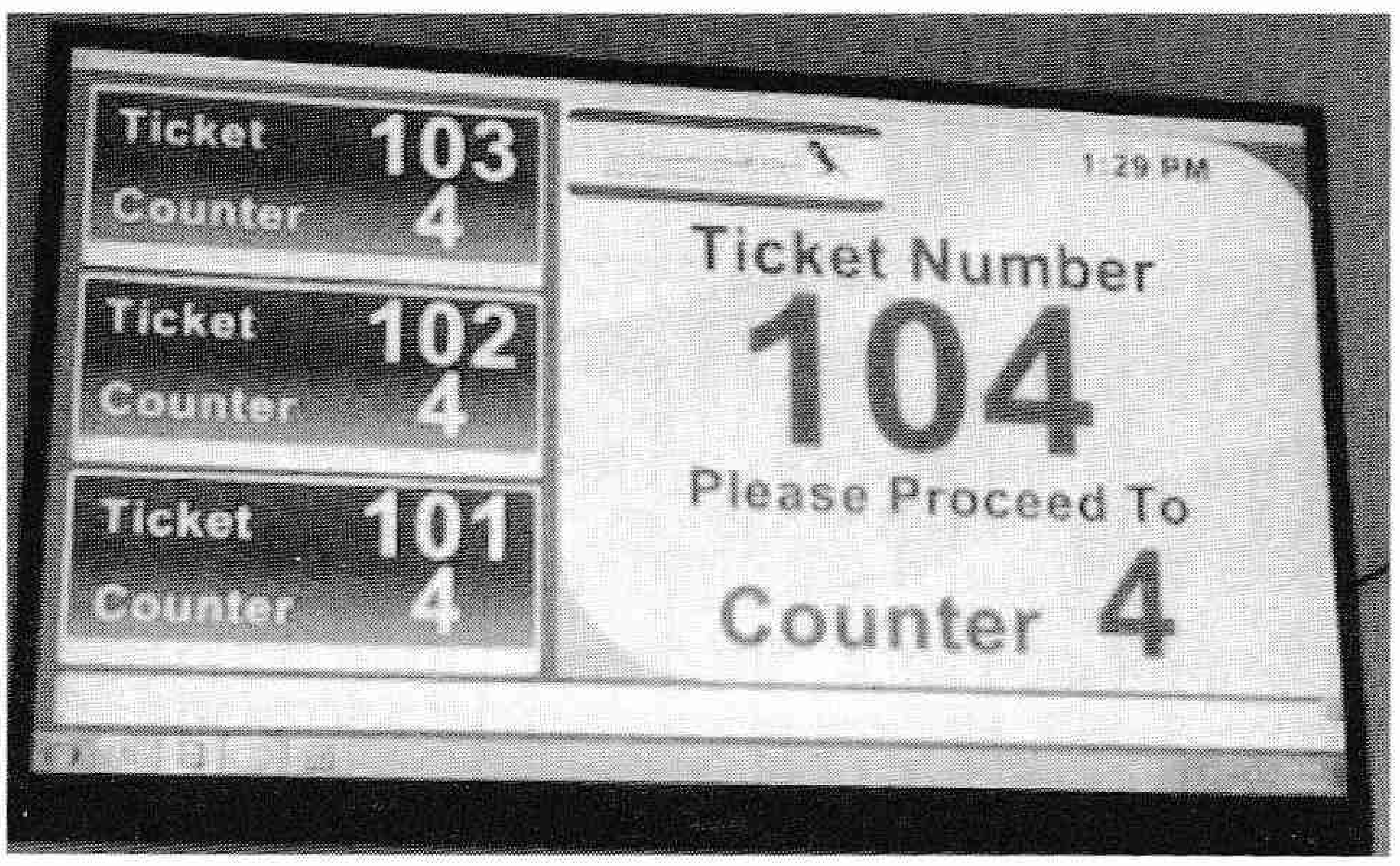
কিউ (Queue)

ইংরেজিতে কিউ (queue) শব্দের অর্থ সারি বা লাইন। যেমন নিচের ছবিটি—



ছবি 5.5

অনেক সময় সফটওয়্যার তৈরির জন্য আমাদেরকে এমন ডেটা স্ট্রাকচার ব্যবহার করতে হয়, যেখানে আগে আসলে আগে পাবেন ভিত্তিতে কাজ করতে হয়। যেমন, আমরা যদি ডাক্তারের চেম্বারের জন্য একটি সফটওয়্যার বানাই, সেখানে ডাক্তারের সাক্ষাৎ পেতে হলে একটি টোকেন ব্যবস্থা বানাতে হবে। টোকেনগুলো কিউ ডেটা স্ট্রাকচারে রাখতে হবে, যেন, যে আগে আসে, সে আগে ডাক্তার দেখাতে পারে। তেমনি রেস্টুরেন্ট, ব্যাংক ইত্যাদি জায়গায়, যেখানেই আমাদের লাইনে দাঁড়াতে হয়, সেখানেই এরকম সফটওয়্যারের ব্যবহার করা যায়, যার কাজ হবে কিউ-এর ধারণা প্রয়োগ করা। তাহলে, সেখানে যারা আসবে, তারা একটি যন্ত্র থেকে টোকেন নিয়ে এক জায়গায় বসে থাকতে পারবে, লাইনে দাঁড়াতে হবে না। ঢাকায় এখন অনেক জায়গায় এমন ব্যবস্থা আছে (ছবি 5.6)।



ছবি 5.6

এরকম দরকারি ও গুরুত্বপূর্ণ সফটওয়্যারের পেছনে যে কিউ ডেটা স্ট্রাকচার ব্যবহার করা হয়, তা কিন্তু মোটেও জটিল কিছু নয়। স্ট্যাকে যেমন যে সবার শেষে আসে, সে সবার আগে চলে যায়, কিউ-এর বেলায় তার উলটো ঘটনা ঘটে, যে সবার আগে আসে, সে সবার আগে চলে যায়। অর্থাৎ First In First Out (FIFO)।

ধরা যাক, একটি কিউতে চারটি সংখ্যা আছে (ছবি 5.7)—

4	5	2	3			

ছবি 5.7

এখন কিউতে নতুন কোনো উপাদান আসলে তার স্থান হবে সবার শেষে (ছবি 5.8)—

4	5	2	3	6		

ছবি 5.8

আর কিউ থেকে কাউকে ডাকা হলে, প্রথমে যে আছে, সে ডাক পাবে (ছবি 5.9)—

4	5	2	3	6		

ছবি 5.9

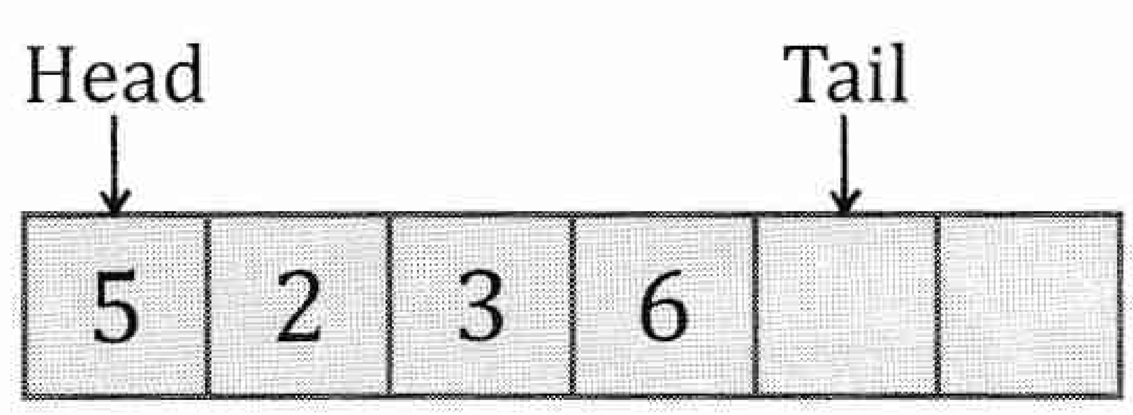
এবং তারপরে চলে যাবে (ছবি 5.10)—

5	2	3	6		

ছবি 5.10

কিউতে নতুন সদস্য যোগ করার কাজকে কম্পিউটার বিজ্ঞানের ভাষায় বলে এনকিউ (enqueue), আর কিউ থেকে কোনো সদস্যকে সরানোর কাজকে বলে ডিকিউ (dequeue)।

কিউ তৈরি করার জন্য একটি অ্যারের সঙ্গে আমাদেরকে দুটি অতিরিক্ত ভ্যারিয়েবল ব্যবহার করতে হবে। একটি ভ্যারিয়েবল দিয়ে আমরা কিউ-এর শুরু কোথায়, আরেকটি ভ্যারিয়েবল দিয়ে কিউ-এর শেষ কোথায়— এ দুটি স্থান শনাক্ত করব। সাধারণত এই দুটি ভ্যারিয়েবলের নাম দেওয়া হয় head ও tail (কেউ কেউ অবশ্য head-এর বদলে front, tail-এর বদলে rear ব্যবহার করে)। কিউতে নতুন কোনো উপাদান এলে তাকে আমরা রাখব tail-এ, আর কিউ থেকে কাউকে সরাতে হলে আমরা head থেকে সরাব (ছবি 5.11)।



ছবি 5.11

ওপরের ছবিতে head-এর মান 0 আর tail-এর মান 4। এখন, আমরা যদি এই কিউতে dequeue() অপারেশন চালাই, তাহলে head-এর মান 1 বাড়িয়ে দিতে হবে।

```
item = q[head];
head = head + 1;
```

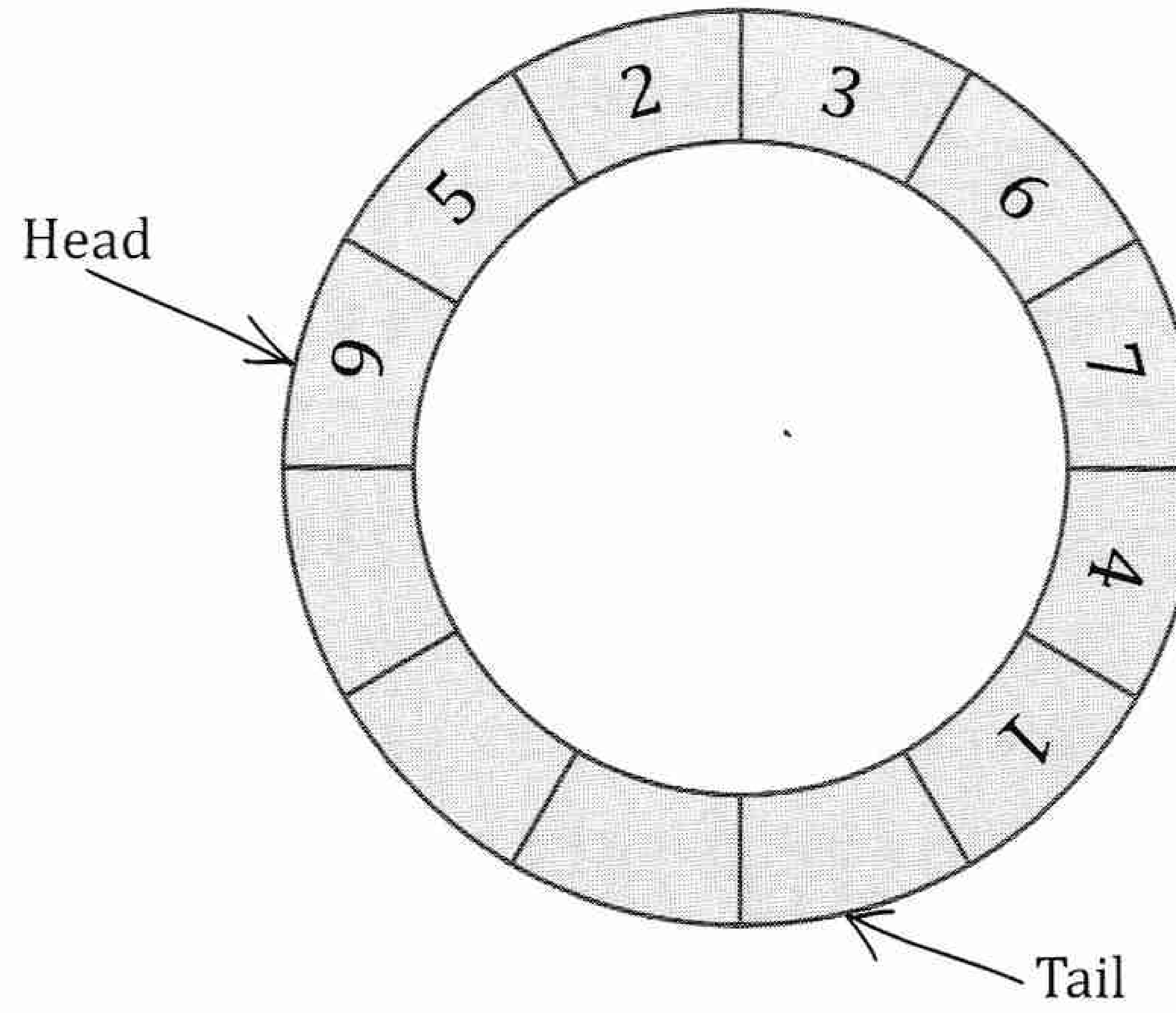
আবার কিউতে যদি enqueue() অপারেশন চালাই, অর্থাৎ নতুন কাউকে যোগ করি, তখন tail-এর মান এক বাড়াতে হবে—

```
q[tail] = new_item;
tail = tail + 1;
```

ওপরের পদ্ধতিতে কাজ করলে enquque() ও dequeue()— দুটো মেথডের কমপ্লেক্সিটিই হবে O(1)। এখন আমরা সহজেই একটি কিউ তৈরি করে ফেলতে পারি। শুরুতে head ও tail-এর মান হবে 0। তারপরে কিউতে ডেটা রাখলে tail-এর মান বাড়বে, কিউ থেকে ডেটা সরিয়ে নিলে head-এর মান বাড়বে। এভাবে কিউতে আমরা বিভিন্ন জিনিস যুক্ত করা কিংবা সরিয়ে নেওয়ার কাজ করতে থাকলে head ও tail-এর মান বেড়েই চলবে। তাই কিউ তৈরি করতে আমাদের বিশাল বড়ো সাইজের একটি অ্যারের প্রয়োজন হবে। আমরা যদি কিউতে একটি জিনিস রাখি, তারপর সেটি সরিয়ে নিই, এরকম 1 লক্ষবার করলে head ও tail –প্রত্যেকের মানই হবে 1 লক্ষ; 1 কোটিবার কাজগুলো করলে প্রত্যেকের মান হবে 1 কোটি, যদিও কিউতে

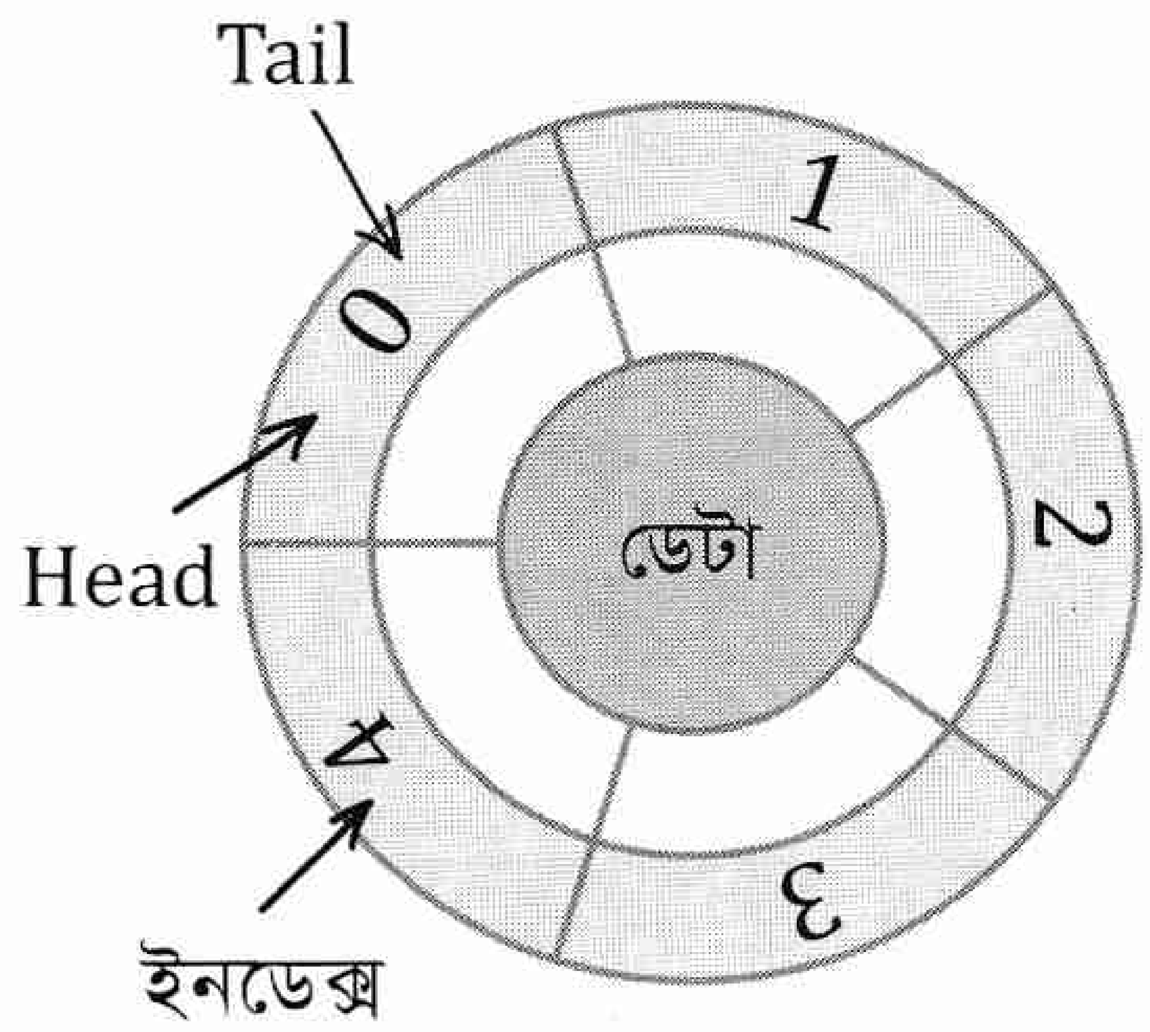
কিন্তু এখন কিছু নেই। এভাবে কিউ তৈরি করা মানে হচ্ছে কম্পিউটারের মেমোরি অপচয় করা, তাই আমরা সেভাবে কাজ করতে যাব না।

এখন, আমরা যখন কোনো কিউ তৈরি করব, তখন আমরা অনুমান করতে পারব যে, কিউতে আসলে সর্বোচ্চ কয়টি ডেটা থাকতে পারে। এটা যদি জাতীয় বিশ্ববিদ্যালয়ের ভর্তি কিউ হয়, তাহলে 10 লক্ষ বা 20 লক্ষ ডেটা থাকতে পারে। আবার যদি ডাক্তারের চেম্বারের কিউ হয়, তাহলে 100 বা 200 ডেটা থাকবে। আবার এটি যদি প্রোগ্রামিং প্রতিযোগিতার সমস্যা হয়, তখন সেখানে কিউতে সর্বোচ্চ কয়টি ডেটা থাকবে তা বলে দেওয়া থাকবে। তাই আমরা যদি সর্বোচ্চ সংখ্যাটি জানি (ধরা যাক সেটি n), তাহলে n সাইজের একটি অ্যারে ব্যবহার করলেই হবে। যখন `tail`-এর মান বাড়তে বাড়তে `n`-এর সমান হয়ে যাবে, তখন সেটিকে আমরা 0 করে দেব। এই কাজটি করার সহজ বুদ্ধি হচ্ছে, `tail = (tail + 1) % n`, তাহলে `(tail + 1)` যদি `n`-এর সমান হয়, তখন `tail`-এর মান `0` হয়ে যাবে (`n`-কে n দিয়ে ভাগ করলে ভাগশেষ `0`)। `head`-এর ক্ষেত্রেও আমরা একই বুদ্ধি প্রয়োগ করব। এভাবে তৈরি করা কিউকে অনেক সময় সার্কুলার (circular) কিউ বলে (ছবি 5.12)।

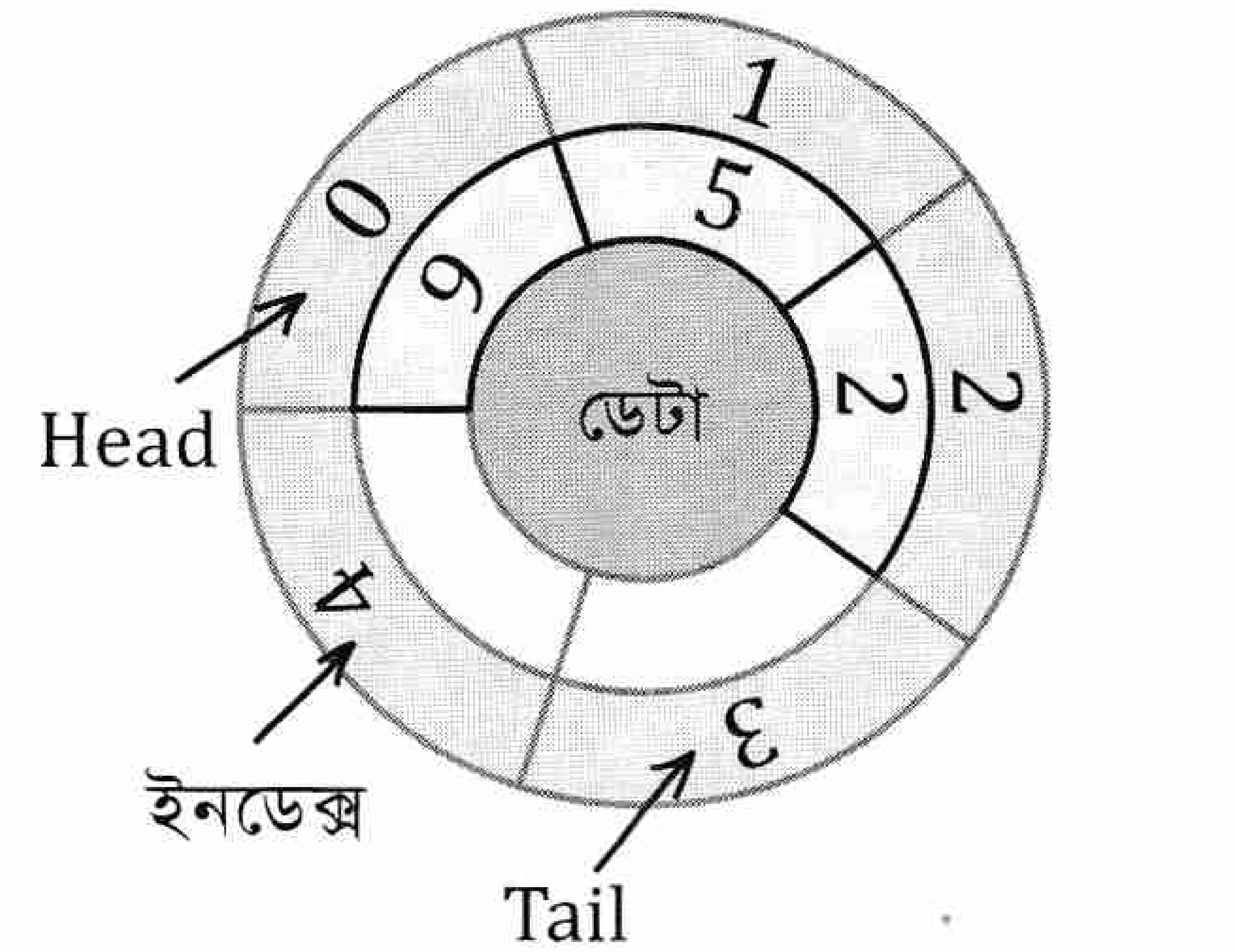


ছবি 5.12

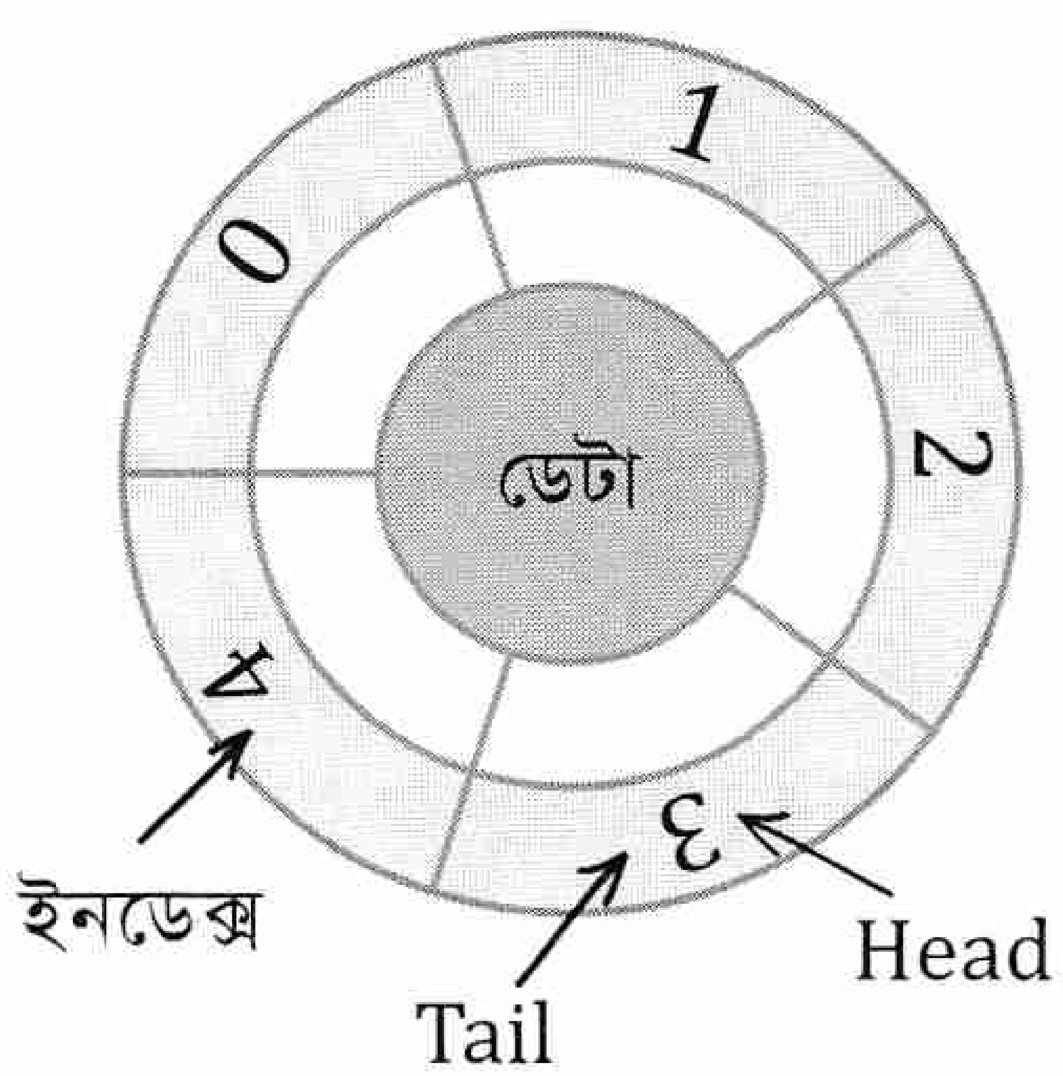
শুরুতে আমাদেরকে `head` ও `tail`-এর মান `0` করে দিতে হবে। কারণ তখন কিউতে কিছুই নেই (ছবি 5.13)। ধরা যাক, কিউতে একে একে তিনটি জিনিস রাখা হলো। তখন প্রতিবার `tail`-এর মান এক করে বেড়ে দাঁড়াবে `3` (ছবি 5.14)। এখন কিউ থেকে আমরা সেই তিনটি জিনিস একে একে সরিয়ে ফেলব। তাহলে `head`-এর মানও তিনবার এক করে বেড়ে হবে `3` (ছবি 5.15)। এখন কিন্তু কিউ খালি। তাহলে আমরা বলতে পারি, যখন `head` ও `tail`-এর মান সমান, তখন কিউ খালি।



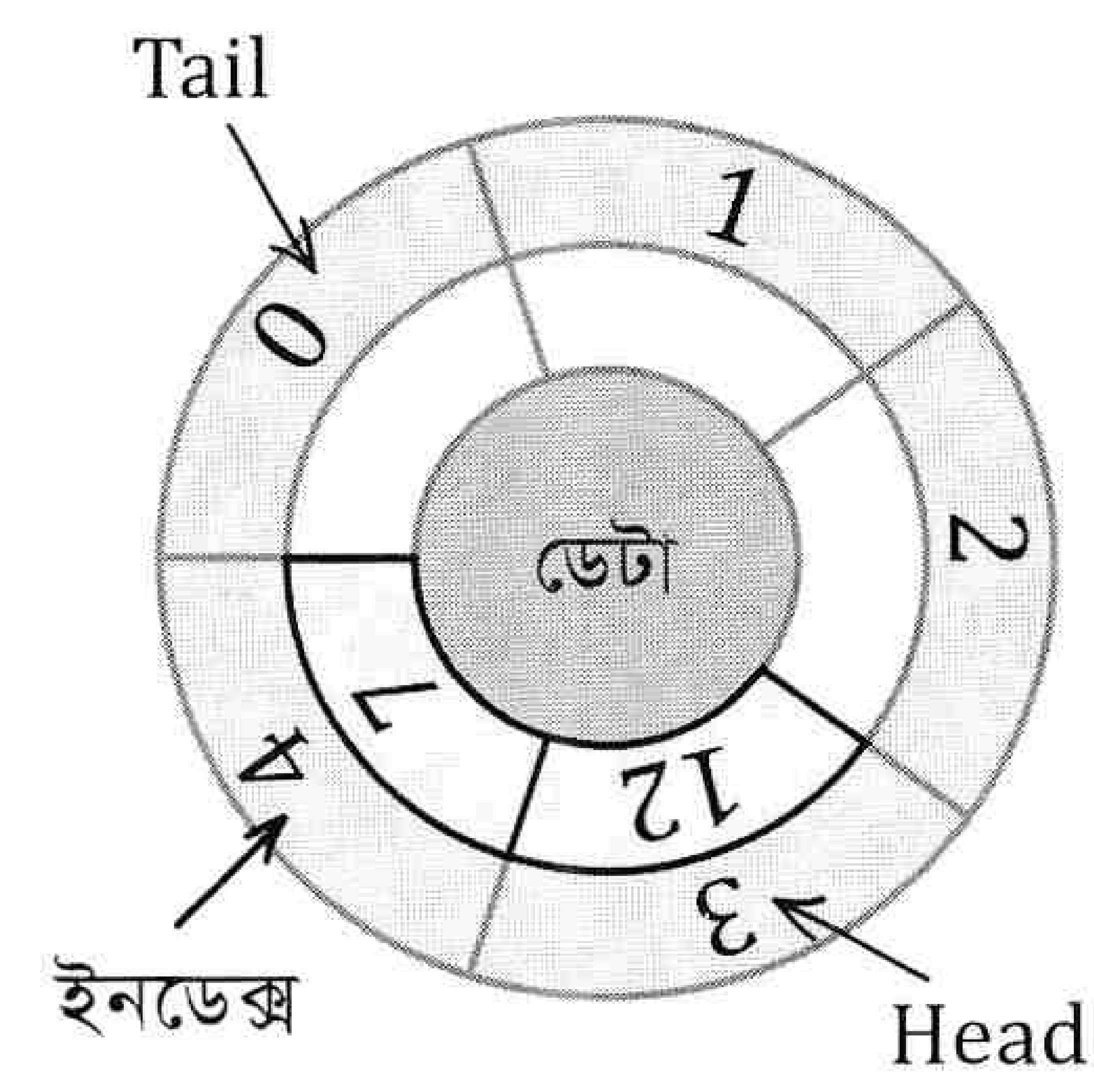
ছবি 5.13: ফাঁকা কিউ



ছবি 5.14: কিউতে 3টি জিনিস রাখাতে tail-এর মান 3 হলো



ছবি 5.15: কিউ থেকে জিনিসগুলো সরিয়ে ফেলায় head-এর মান 3 হলো



ছবি 5.16 কিউতে আরে 2টি জিনিস রাখলে tail-এর মান 0 হবে

এখন ধরা যাক, কিউ-এর সাইজ হচ্ছে 5। তাহলে আমরা কিউতে সর্বোচ্চ পাঁচটি জিনিস রাখতে পারব। আমরা কখন বুঝব যে কিউতে কোনো জায়গা খালি নেই? আগের উদাহরণে যখন তিনটি জিনিস কিউতে রেখে আবার সরিয়ে নিয়েছিলাম, তখন `head` ও `tail` উভয় ভ্যারিয়েবলের মান দাঁড়িয়েছিল 3। এখন কিউতে কোনো কিছু নেই। আমরা যদি কিউতে পাঁচটি জিনিস রাখি, তাহলে প্রতিবার `tail`-এর মান বাড়বে এভাবে—

প্রথম জিনিস রাখার পরে `tail`-এর মান 1 বেড়ে হবে 4।

দ্বিতীয় জিনিস রাখার পরে `tail`-এর মান 5 হওয়ার কথা, কিন্তু অ্যারেতে যেহেতু সর্বোচ্চ 5টি উপাদান রাখা যাবে, তাই এর ইনডেক্সের সর্বোচ্চ মান হবে 4। তাই আমরা `tail`-এর মান বাড়াব এভাবে : `tail = (tail + 1) % 5`। তাই এখানে `tail`-এর মান হবে `0 (5 % 5` এর মান `0)`।

এখন আরো তিনটি জিনিস রাখলে `tail`-এর মান বেড়ে হবে 3। এখন কিউ পরিপূর্ণ— আর কোনো জায়গা খালি নেই। আর `head` ও `tail` উভয়েরই মান সমান `(3)`। তাহলে তো একটু

ঝামেলা হয়ে গেল, head ও tail সমান হলে কিউ খালি নাকি পরিপূর্ণ? এই সমস্যা সমাধানের উপায় কী? তোমরা একটু চিন্তাভাবনা করো। আশা করি, তোমাদের অনেকেই এই সমস্যা সমাধানের জন্য একাধিক উপায় খুঁজে বের করবে।

ওপরের সমস্যা সমাধান করার জন্য আমরা যেটি করতে পারি, কিউতে যদি সর্বোচ্চ 5টি উপাদান রাখা প্রয়োজন হয়, তাহলে অ্যারের সাইজ এক বেশি নেব। আর তারপর, কিউতে ডেটা রাখার সময় আমরা দেখব যে, tail + 1 যদি head-এর সময় হয়, তাহলে আর সেই ঘরে ডেটা রাখব না, বরং বলে দেব যে, কিউতে আর জায়গা খালি নেই। তাহলে আমরা আগের উদাহরণে ফেরত যাই, যেখানে কিউতে তিনটি জিনিস রাখা এবং সেগুলো সরিয়ে নেওয়ার পরে head ও tail উভয়ের মান 3 হয়েছিল। এখন আমরা আবার পাঁচটি জিনিস রাখব। প্রতিবার tail-এর মান বাড়াব এভাবে : tail = (tail + 1) % 6। আমরা কিন্তু 6 দিয়ে মড করছি, 5 দিয়ে নয়। তাহলে পাঁচটি জিনিস রাখলে tail-এর মান দাঁড়াবে 2 (4, 5, 0, 1, 2)। এখন যদি আমরা কিউতে কিছু রাখতে যাই, তাহলে tail + 1 হবে 3, যা head-এর সমান। তার মানে কিউ পরিপূর্ণ, আমরা নতুন কিছু রাখতে পারব না। তাই আমরা যেই বুদ্ধিটি করলাম, তা হচ্ছে, অ্যারের সাইজ এক বেশি নেওয়া এবং head-এর আগের ঘরটি খালি রাখা।

অনেক তো আলোচনা হলো। আমি এখন কিউ-এর কোড লিখে দেখাব। তবে তার আগে তোমাদের একটি কাজ করতে হবে; ঠান্ডা মাথায় কিউ অধ্যায়টি শুরু থেকে এই পর্যন্ত আবার পড়তে হবে। তারপরে তোমরা নিচের প্রোগ্রাম দেখবে।

```
#include <stdio.h>

#define Q_SIZE 5

typedef struct {
    // আমরা data  অ্যারের সাইজ দিচ্ছি Q_SIZE + 1। কারণ আগেই ব্যাখ্যা
    // করা হয়েছে। তাই head ও tail-এর মান যখনই এক বাড়াবে, তখন
    // কিন্তু Q_SIZE দিয়ে নয়, বরং Q_SIZE+1 দিয়ে মড করতে হবে।
    int data[Q_SIZE + 1];
    int head, tail;
} Queue;

void enqueue(Queue *q, int item)
{
    if ((q->tail + 1) % (Q_SIZE + 1) == q->head) {
        // এখানে কিন্তু q->tail + 1 == q->head পরীক্ষা করলে হবে না।
        // তোমরা নিজেরা কোড পরিবর্তন করে দেখতে পারো। কেন হবে না, সেটি
```

```
        // চিন্তা করে বের করো।
        printf("Queue is full\n");
        return;
    }
    q->data[q->tail] = item;
    q->tail = (q->tail + 1) % (Q_SIZE + 1);
}

int dequeue(Queue *q) {
    int item;

    if (q->tail == q->head) {
        printf("Queue is empty\n");
        return -1;
    }

    item = q->data[q->head];

    q->head = (q->head + 1) % (Q_SIZE + 1);

    return item;
}


int main()
{
    Queue my_q;
    int item;

    // শুরুতে my_q-এর head ও tail-এর মান 0 করে দিই।
    my_q.head = 0;
    my_q.tail = 0;

    enqueue(&my_q, 1);
    printf("tail = %d\n", my_q.tail);
    enqueue(&my_q, 2);
    printf("tail = %d\n", my_q.tail);
    enqueue(&my_q, 3);
    printf("tail = %d\n", my_q.tail);
    enqueue(&my_q, 4);
    printf("tail = %d\n", my_q.tail);
```

```
    enqueue(&my_q, 5);
    printf("tail = %d\n", my_q.tail);
    enqueue(&my_q, 6);
    printf("tail = %d\n", my_q.tail);

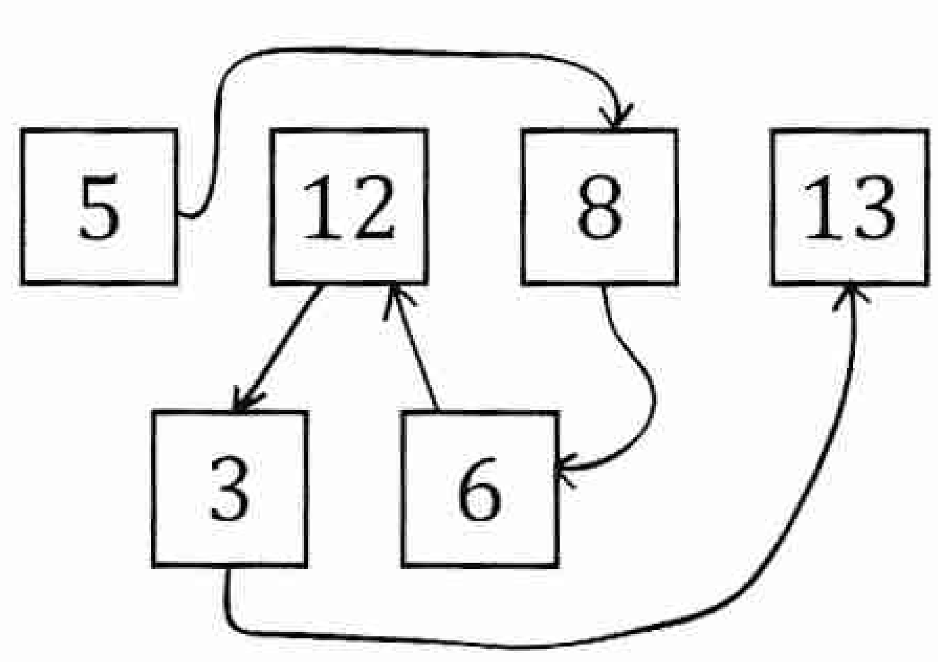
    printf("beginning head = %d\n", my_q.head);
    item = dequeue(&my_q);
    printf("item = %d, head = %d\n", item, my_q.head);
    item = dequeue(&my_q);
    printf("item = %d, head = %d\n", item, my_q.head);
    item = dequeue(&my_q);
    printf("item = %d, head = %d\n", item, my_q.head);
    item = dequeue(&my_q);
    printf("item = %d, head = %d\n", item, my_q.head);
    item = dequeue(&my_q);
    printf("item = %d, head = %d\n", item, my_q.head);
    item = dequeue(&my_q);
    printf("item = %d, head = %d\n", item, my_q.head);
}
```

ওপরের কোড পড়ে বোঝার চেষ্টা করতে হবে এবং তারপর কোড রান করে দেখতে হবে।

অধ্যায় ৬ : লিংকড লিস্ট (Linked List)

লিংকড লিস্টের ধারণা

আমরা যখন অ্যারের ভেতরে কোনো ডেটা রাখি, তখন প্রথম উপাদানটি থাকে অ্যারের প্রথম স্থানে (যেটির ইনডেক্স 0), দ্বিতীয় উপাদানটি থাকে অ্যারের দ্বিতীয় স্থানে (ইনডেক্স 1), তৃতীয় উপাদানটি থাকে তৃতীয় স্থানে (ইনডেক্স 2)– সবগুলো উপাদান এরকম ধারাবাহিকভাবে থাকে। এই অধ্যায়ে আমরা জানব একটি নতুন ডেটা স্ট্রাকচার সম্পর্কে, যার নাম লিংকড লিস্ট (linked list)। এর মূল বৈশিষ্ট্য হচ্ছে, ডেটাগুলো ধারাবাহিকভাবে থাকে না, বরং একেক ডেটা একেক জায়গায় থাকে। তাই সরাসরি ইনডেক্স ব্যবহার করে ডেটা পাওয়া যায় না। কিন্তু একটি ডেটার সঙ্গে তার পরের ডেটার একটি সংযোগ বা লিংক থাকে। তাই এই ডেটা স্ট্রাকচারকে বলা হয় লিংকওয়ালা লিস্ট বা লিংকড লিস্ট। লিংকড লিস্টে ডেটাগুলো অনেকটা ছবি 6.1-এর মতো থাকে।



ছবি 6.1

আমরা দেখতে পাচ্ছি, প্রতিটি ডেটার সঙ্গে আরেকটি তথ্য থাকে, যেটি হচ্ছে, তার পরের ডেটা কোথায় আছে তার ঠিকানা। সেটি ছবিতে তিরচিহ্ন দিয়ে বোঝানো হয়েছে। তাই, 5-এর পরের ডেটাটি হচ্ছে 8, তারপর 6, তারপর 12, তারপর 3, তারপর 13। 13-থেকে কোনো তিরচিহ্ন বের হচ্ছে না, তার অর্থ হচ্ছে 13-এর পরে আর কোনো ডেটা নেই। তাহলে আমরা দেখলাম, এখানে একটি ডেটাতে কেবল সেই ডেটা রাখলেই চলে না, বরং তার পরের ডেটা কোথায় আছে, সেই তথ্যও রাখতে হয়। তাই লিংকড লিস্টের প্রতিটি উপাদানের আসলে দুটি অংশ থাকে। প্রথমে অংশে থাকে ডেটা। দ্বিতীয় অংশে থাকে, তার পরের উপাদান কোথায় আছে, সেই ঠিকানা (ছবি 6.2)।



ছবি 6.2

লিংকড লিস্টের একটি উপাদানকে বলা হয় নোড (node)। আর প্রথম নোডের ঠিকানা রাখার জন্য আলাদা একটি নোড (সেই নোডকে সাধারণত head বলা হয়) ব্যবহার করা হয়, যেখানে লিংকড লিস্টের প্রথম বা শুরুর নোডের ঠিকানা থাকে। এখন আমি দুটি অ্যারে ব্যবহার করে লিংকড লিস্টের ধারণাটি আরেকটু সহজভাবে বোঝানোর চেষ্টা করব। নিচের ছবিতে, আমরা Data নামে একটি অ্যারে এবং Next নামে আরেকটি অ্যারে ব্যবহার করেছি। লিংকড লিস্টের প্রথম নোডটি আছে প্রথম সারিতে, অর্থাৎ এর ডেটা আছে `Data[0]`-তে আর তার পরের নোডের ঠিকানা আছে `Next[0]`-তে।

Index	Data	Next
0	5	3
1		
2	12	9
3	8	12
4		
5		
6		
7	13	15
8		
9	3	7
10		
11		
12	6	2
13		
14		
15	9	
16		

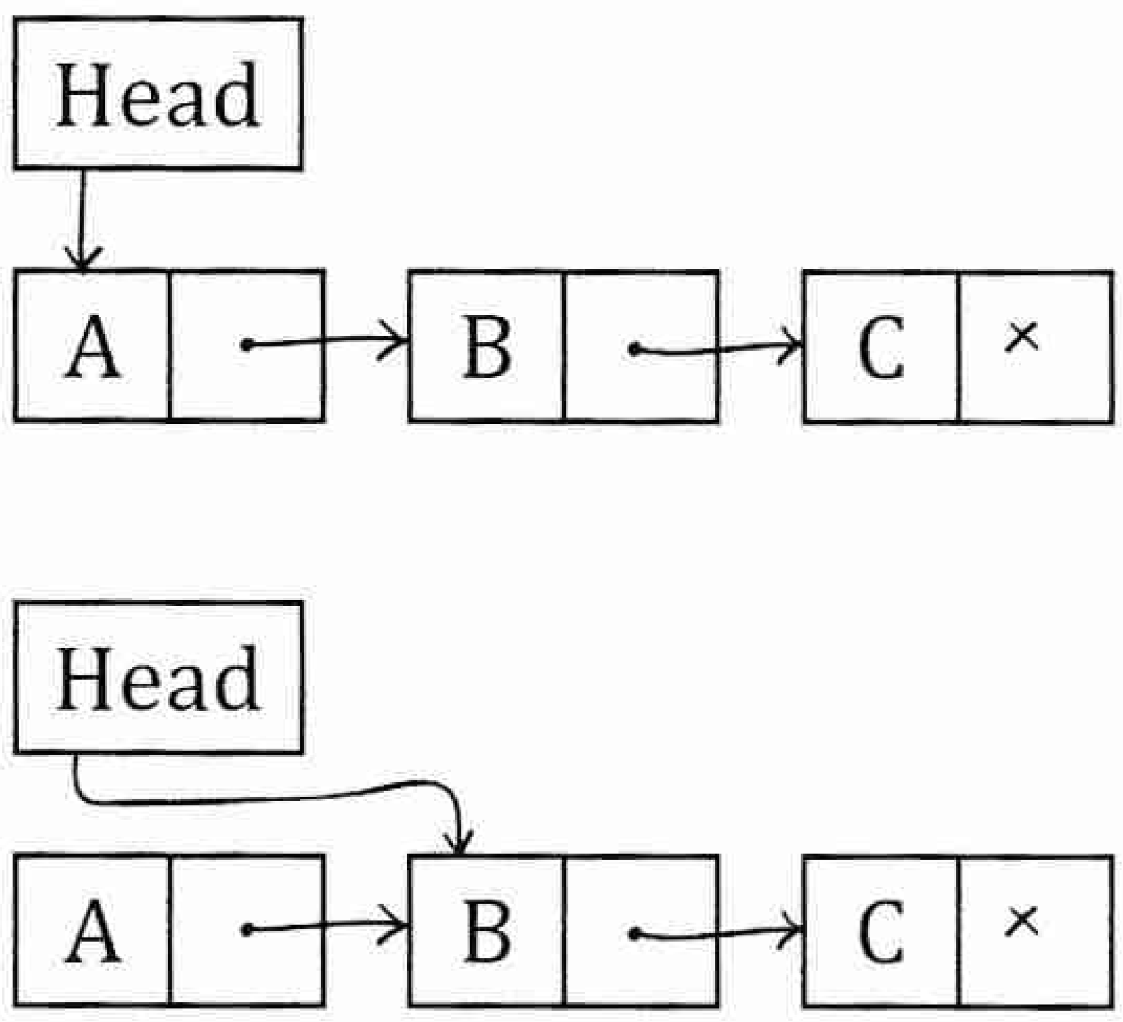
এখন প্রশ্ন হচ্ছে, দ্বিতীয় উপাদানটি কোথায় আছে? সেটির তথ্য আছে Next নামে যেই অ্যারে তৈরি করেছি, সেখানে। Data[0]-এর পরবর্তী উপাদানটির ঠিকানা আছে Next[0]-তে আর Next[0]-এর মান হচ্ছে 3। তাই আমরা পরবর্তী নোডটি পাব 3 নম্বর ইনডেক্সে। ওই নোডের ডেটা থাকবে Data[3]-এ। লিংকড লিস্টের তৃতীয় উপাদানটি আমরা পাব Next[3]-এর মান যত, তত নম্বর ইনডেক্সে, অর্থাৎ 12 নম্বর ইনডেক্সে। সেখানে আছে Data[12] অর্থাৎ, 6। তার পরের উপাদানটি আছে Next[12] নম্বর ইনডেক্স, অর্থাৎ 2 নম্বর ইনডেক্সে। সেখানে আছে Data[2] বা 12। তার পরের নোডটি আমরা পাব Next[2] ইনডেক্সে, তাই সেটি হচ্ছে Data[9] (যার মান হচ্ছে 3)। তারপরের উপাদানটি পাওয়া যাবে Next[9] বা 7 নম্বর ইনডেক্সে। তারপরের উপাদানটি আছে Next[7] বা 15 নম্বর ইনডেক্সে, সেটি হচ্ছে 9 (Data[15])। এটিই লিস্টের শেষ উপাদান কারণ Next[15]-তে কিছু নেই। তাহলে আমরা যদি লিংকড লিস্টটির নোডগুলোর ডেটা শুরু থেকে ক্রমানুসারে লিখি, সেগুলো হবে, 5, 8, 6, 12, 3, 13, 9। কিন্তু আমরা দেখেছি, এগুলো লিস্টের মধ্যে পরপর নেই। তাতে কোনো সমস্যা নেই, কারণ প্রতিটি উপাদানেই পরের উপাদানটির লিংক আছে, এ কারণেই একে বলে লিংকড লিস্ট।

অ্যারে ব্যবহার করার সময়, আমরা যদি জানি যে, কত নম্বর ডেটা আমাদের দরকার, সেটি আমরা সহজেই ইনডেক্স ব্যবহার করে বের করে ফেলতে পারি। ধরা যাক, অ্যারের নাম হচ্ছে A। তাহলে প্রথম উপাদানটি আমরা পাব A[0]-তে, দ্বিতীয় উপাদানটি আমরা পাব A[1]-এ, দশম উপাদানটি আমরা পাব A[9]-এ। এগুলো হচ্ছে $O(1)$ অপারেশন। আর লিংকড লিস্টের বেলাতে, আমাদেরকে যদি দশম উপাদানটি বের করতে বলা হয়, তাহলে আমরা প্রথমে দেখব head নোডটি কোথায় নির্দেশ করছে। তারপর সেখান থেকে পরবর্তী নোডের ঠিকানা নিয়ে যেতে যেতে আমরা দশম উপাদান বের করে ফেলতে পারব। n-তম উপাদান বের করার জন্য আমাদেরকে একটি লুপ ব্যবহার করে nসংখ্যক বার কাজ করতে হবে। তাই এটির কমপ্লেক্সিটি হচ্ছে $O(n)$।

এখন আমরা দেখব, কীভাবে লিংকড লিস্ট থেকে নোড বাদ দিতে পারি এবং কীভাবে নতুন নোড যোগ করতে পারি।

লিংকড লিস্ট থেকে কোনো নোড বাদ দেওয়া অর্থাৎ ডিলিট করার সময় প্রথমে আমাদেরকে সেই নোডটি খুঁজে বের করতে হবে। তবে ব্যতিক্রম হচ্ছে, যখন আমরা প্রথম নোডটি বাদ দেব। সে ক্ষেত্রে আমরা জানি যে, তার ঠিকানা head নোডে আছে। তাই head নোডে যদি A নোডের ঠিকানা থাকে, তাহলে প্রথম নোডটি হচ্ছে A আর দ্বিতীয় নোডের ঠিকানা আছে A নোডের মধ্যে। আমরা করব কী, A নোডের মধ্যে যে অংশে পরের নোডের ঠিকানা আছে, সেই অংশটুকু head

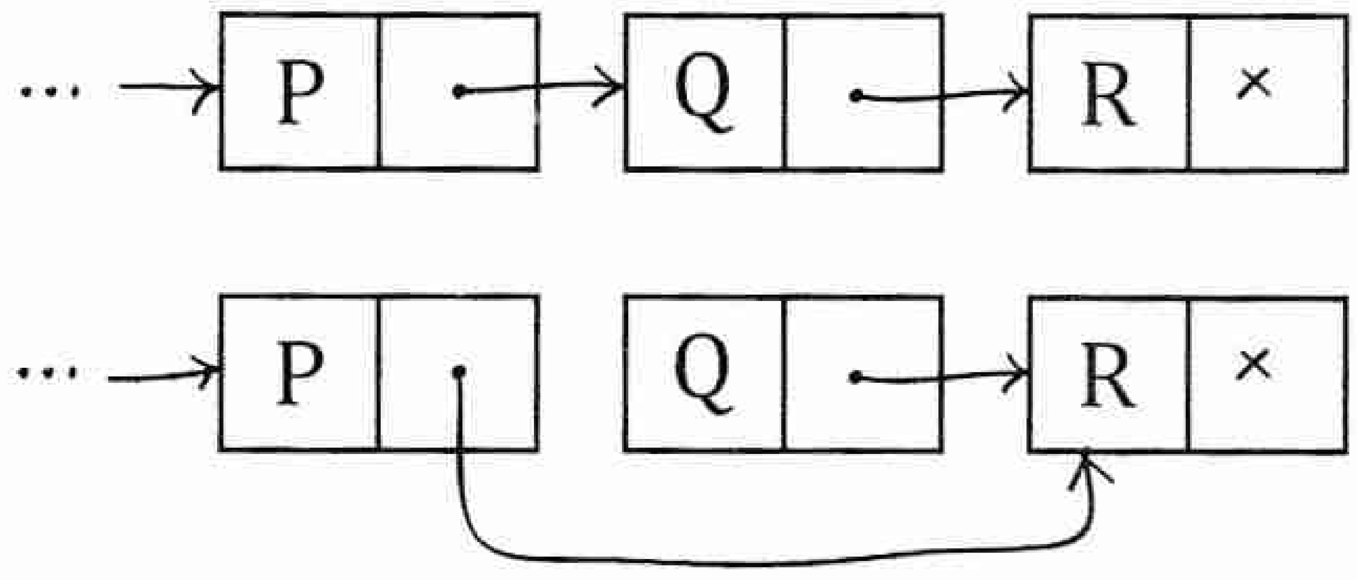
নোডের মধ্যে কপি করে দেব। তাহলে head নোড এখন A–এর পরবর্তী নোডকে নির্দেশ করবে (ছবি 6.3)।



ছবি 6.3

ছবি 6.3-এ আমরা দেখতে পাচ্ছি, head এবং A, দুটি নোডই B নোডকে নির্দেশ করছে। কিন্তু আমরা তো লিংকড লিস্ট শুরু করব head যাকে নির্দেশ করে সেখান থেকে, আর সেটি এখন হচ্ছে B। তাই আমাদের লিংকড লিস্ট থেকে A হারিয়ে গেল বা মুছে গেল। লিংকড লিস্টের শুরুর নোড ডিলিট করা হচ্ছে $O(1)$ অপারেশন বা এর টাইম কমপ্লেক্সিটি হচ্ছে $O(1)$।

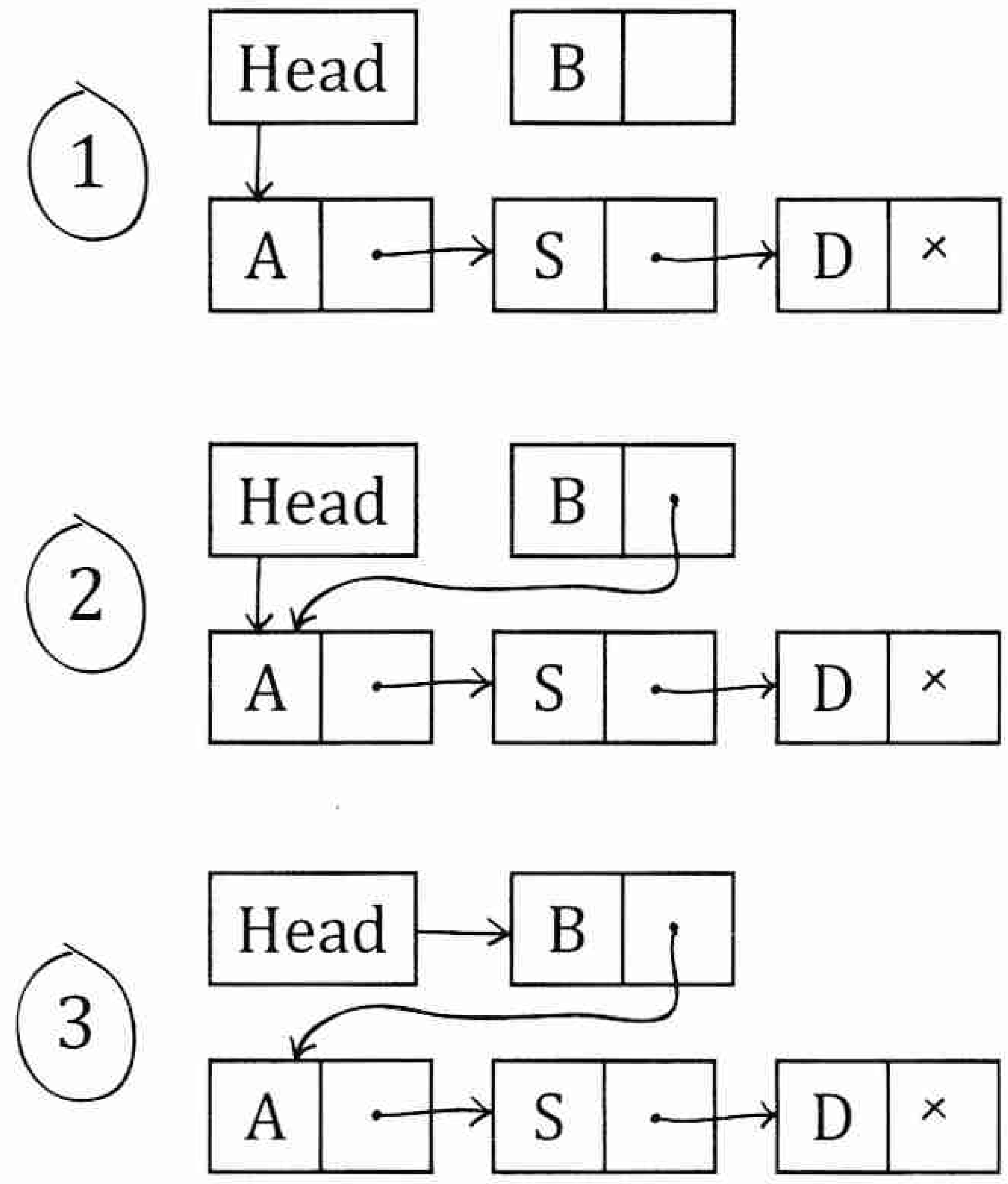
আবার, আমরা যদি লিংকড লিস্টের ভেতরে কোনো একটি নোড Q মুছে ফেলতে চাই বা বাদ দিতে চাই, তাহলে প্রথমে আমাদেরকে Q এবং তার পূর্ববর্তী নোড (ছবিতে P) খুঁজে বের করতে হবে। তারপর Q–এর মধ্যে তার পরবর্তী নোড R–এর যে ঠিকানা আছে, সেটি P নোডের ঠিকানা অংশে কপি করে ফেলব। তাহলে এখন লিংকড লিস্টে P–এর পরে R থাকবে আর Q হারিয়ে যাবে (ছবি 6.4)। এই কাজটি করার জন্য যেহেতু Q নোডটি খুঁজে বের করার প্রয়োজন হয়েছে, তাই এটি $O(n)$ অপারেশন, কারণ Q যদি n-তম অবস্থানে থাকে, তাহলে সেটি খুঁজে বের করতে $O(n)$ সময় লাগবে। তারপর মুছতে লাগবে $O(1)$ সময়। তাই কমপ্লেক্সিটি হচ্ছে $O(n)$।



ছবি 6.4

এখন আমরা দেখব, কীভাবে লিংকড লিস্টে নতুন নোড যোগ করতে হয়। যদি আমাদের বলা হয়, শুরুতে কোনো নোড যোগ করতে, তাহলে আমরা প্রথমে নতুন নোডের ভেতরে head নোড যাকে নির্দেশ করে, তার ঠিকানা রাখব। তারপরে head নোডের ভেতরে নতুন নোডের ঠিকানা রেখে দেব।

ছবি 6.5-এ যেমন আমরা নতুন একটি নোড B-কে লিংকড লিস্টের শুরুতে যুক্ত করতে চাইছি।



ছবি 6.5

প্রথমে B-এর ভেতরে A-এর ঠিকানা (যেটি head-এর মধ্যে আছে) রাখতে হবে। তারপরে head-এ B-এর ঠিকানা রাখতে হবে। এই কাজের কমপ্লেক্সিটি হচ্ছে $O(1)$। এখন আমাদের যদি বলা হয়, লিংকড লিস্টের একটি নির্দিষ্ট নোডের পরে একটি নোড যুক্ত করতে হবে, তখন সেই নির্দিষ্ট নোডটি আমাদেরকে খুঁজে বের করতে হবে। তারপরে নতুন নোড যুক্ত করতে হবে। তাই এর কমপ্লেক্সিটি হবে $O(n)$।

সি দিয়ে লিংকড লিস্ট তৈরি

এখন আমরা সি দিয়ে লিংকড লিস্ট তৈরি করব। আগেই জানিয়ে দিতে চাই যে, কারো কারো প্রথমবার পড়ে বুঝতে সমস্যা হতে পারে। সমস্যা হলে দুটি সমাধান আছে– প্রথমত খাতা-কলম নিয়ে বসে পরা এবং কোন লাইন কী করছে তার ছবি এঁকে ফেলা। এর পরও বুঝতে সমস্যা হলে কোনো বন্ধুর সাহায্য নিতে হবে। আমার ব্যক্তিগত অভিজ্ঞতা বলে যে, অনলাইনের চেয়ে সামনাসামনি বন্ধুর সাহায্য অনেক বেশি কার্যকর। আরেকটি বিষয় হচ্ছে, লিংকড লিস্টের কোড বোঝার জন্য, তোমাদের পয়েন্টার ও স্ট্রাকচার সম্পর্কে পরিষ্কার ধারণা থাকতে হবে। সেজন্য তোমরা কম্পিউটার প্রোগ্রামিং ২য় খণ্ড বইটি পড়তে পারো।

প্রথমেই নোড-এর জন্য একটি স্ট্রাকচার তৈরি করতে হবে। আমরা ধরে নিচ্ছি যে, লিংকড লিস্টে ইন্টিজার টাইপের ডেটা নিয়ে কাজ করা হবে। তোমাদের যদি অন্য টাইপের ডেটার জন্য লিংকড লিস্ট তৈরি করতে হয়, তাহলে int-এর বদলে সেই টাইপের ভ্যারিয়েবল ডিক্লেয়ার করবে।

```
struct node {
    int data;
    struct node *next;
};
```

এখন লক্ষ করো, node নামে আমরা একটি স্ট্রাকচার তৈরি করেছি, যার প্রথম উপাদান হচ্ছে একটি ইন্টিজার (data), আর দ্বিতীয় উপাদান হচ্ছে এই node নামক স্ট্রাকচারের পয়েন্টার, যেন সেটি লিংকড লিস্টের এই নোডের পরের নোডকে নির্দেশ করতে পারে, বা পরের নোডের ঠিকানা রাখতে পারে। তাই এই next নামক পয়েন্টারের টাইপ হচ্ছে struct node।

আমরা এক কাজ করি, typedef ব্যবহার করে struct node-কে Node নাম দিয়ে দিই, তাহলে আমাদের বারবার struct node লেখা লাগবে না, বরং এর বদলে Node লিখলেই হবে।

```
typedef struct node Node;

struct node {
    int data;
    Node *next;
};
```

এখন আমরা যদি লিংকড লিস্ট তৈরি করতে চাই, তাহলে আমাদের আরেকটি পয়েন্টার প্রয়োজন হবে, যার নাম হচ্ছে head। head-এর কাজ হচ্ছে লিংকড লিস্টের প্রথম নোডটি নির্দেশ করা। যখন লিংকড লিস্ট তৈরি করব, তখন শুরুতে তো কোনো নোড নেই, তাই head-এ আমরা NULL রেখে দেব, তার মানে head কাউকে নির্দেশ করছে না।

```
Node *head;
```

লিংকড লিস্ট নিয়ে কাজ করার সময় আমাদের প্রায়ই একটি কাজ করতে হবে, সেটি হচ্ছে নোড তৈরি করা। ধরা যাক লিংকড লিস্টে কোনো ডেটা যুক্ত করতে চাই, তাহলে সেই ডেটা নিয়ে আমাদেরকে একটি নোড তৈরি করে সেই নোডকে লিংকড লিস্টে যুক্ত করতে হবে, সরাসরি তো আর ডেটা যুক্ত করা যাবে না। তাই নোড তৈরি করার জন্য একটি ফাংশন লিখে ফেলি।

```
/*
   ফাংশনটি প্যারামিটার হিসেবে একটি ডেটা (int item) ও Node টাইপের পয়েন্টার
   (Node *next) নেবে। ফাংশনটি একটি Node টাইপের পয়েন্টার রিটার্ন করবে।
*/
Node *create_node(int item, Node *next)
{
    /*
      প্রথমে Node-এর যেই সাইজ, সেই পরিমাণ মেমোরি অ্যালোকেট করি। তারপরে
      সেটিকে Node *-এ টাইপকাস্ট করে new_node-এ (যা Node টাইপের
      পয়েন্টার) অ্যাসাইন করি। এই কাজটি করার ফলে, new_node পয়েন্টারটি
      যেখানে নির্দেশ করছে, সেখানে একটি নোড তৈরি করার প্রয়োজনীয় মেমোরি
      বরাদ্দ করা হলো।
    */
    Node *new_node = (Node *) malloc(sizeof(Node));
    if (new_node == NULL) {
        // মেমোরি অ্যালোকেট করার সময় কোনো ত্রুটি হলে new_node-এ NULL
        // অ্যাসাইন হবে।. তাই আমরা সেটি পরীক্ষা করে নিচ্ছি।
        printf("Error! Could Not Create A New Node\n");
        exit(1);
    }

    // এখন new_node-এর ডেটা অংশে item এবং next অংশে next রাখছি।
    // new_node যেহেতু একটি স্ট্রাকচার নয়, বরং স্ট্রাকচারের পয়েন্টার, তাই
    // আমরা new_node.data না লিখে new_node->data এভাবে লিখছি।
    // next-এর বেলাতেও একই কথা প্রযোজ্য।
    new_node->data = item;
    new_node->next = next;

    // new_node (যা কিনা একটি Node *) রিটার্ন করছি।
    return new_node;
}

int main() {

    Node *n;

    // একটি নোড তৈরি করি, যার ডেটা অংশে থাকবে 10 আর next হবে NULL।
    n = create_node(10, NULL);
```

```
    // n নোডটি ঠিকমতো তৈরি হলো কি না, সেটি পরীক্ষা করার জন্য তার ডেটা অংশ
    // প্রিন্ট করি।
    printf("data = %d\n", n->data);

    return 0;
}
```

এখন আমরা কোনো লিংকড লিস্টের শুরুতে ডেটা যুক্ত করব। আমরা জানি যে, লিংকড লিস্টের প্রথম নোডটির ঠিকানা থাকে head-এর কাছে। এখন আমরা যদি লিংকড লিস্টের শুরুতে নতুন নোড যুক্ত করি, তাহলে head-এর কাছে নতুন নোডের ঠিকানা থাকবে। আর head আগে যেই নোডকে নির্দেশ করত, নতুন নোডটি তার পরবর্তী নোড হিসেবে সেই নোডকে নির্দেশ করবে।

```
Node *prepend(Node *head, int item)
{
    // একটি নোড তৈরি করতে হবে যার ডেটা হচ্ছে item এবং তার পরবর্তী নোড
    // হচ্ছে head যাকে নির্দেশ করছে, সেই নোড, তাই দ্বিতীয় আর্গুমেন্টে head
    // পাঠাচ্ছি।
    Node *new_node = create_node(item, head);

    // new_node-কে আমরা রিটার্ন করছি। এটি হচ্ছে বর্তমান লিংকড লিস্টের প্রথম
    // নোড, তাই head পয়েন্টারটি new_node-কে নির্দেশ করবে। সেজন্য আমরা
    // এই ফাংশন থেকে যা রিটার্ন করব, তা head-এ অ্যাসাইন করব।
    return new_node;
}

int main()
{
    Node *n1, *n2, *head, *n3;

    n1 = create_node(10, NULL);
    head = n1;

    // prepend() ফাংশন যা রিটার্ন করবে, তা head-এ অ্যাসাইন করতে হবে।
    head = prepend(head, 20);

    n2 = head;
    printf("first data = %d\n", n2->data);
```

```
    n3 = n2->next;
    printf("second data = %d\n", n3->data);

    return 0;
}
```

ওপরের প্রোগ্রামটি ভালোভাবে লক্ষ করো ও টাইপ করে কম্পাইল ও রান করে দেখো।

আর আমরা যদি লিংকড লিস্টের শেষে একটি নতুন নোড যুক্ত করতে চাই, তাহলে আমাদের কী করতে হবে? আমরা জানি, প্রথম নোডটির জন্য আমাদের head পয়েন্টার রয়েছে, যা লিংকড লিস্টের প্রথম নোডকে নির্দেশ করে। কিন্তু শেষ নোড নির্দেশ করার তো কোনো পয়েন্টার নেই। তাহলে শেষ নোড খুঁজে পাওয়ার বুদ্ধি কী? আমাদেরকে একটি লুপ চালিয়ে শেষ নোডে চলে যেতে হবে। তারপর সেখানে নতুন নোড যুক্ত করার কাজটি করতে হবে। এ কারণে prepend() ফাংশনের কমপ্লেক্সিটি $O(1)$ হলেও এখন যে append() ফাংশন তৈরি করব, তার কমপ্লেক্সিটি হচ্ছে $O(n)$। উল্লেখ্য যে, prepend (উচ্চারণ প্রিপেন্ড) শব্দের অর্থ শুরুতে যুক্ত করা, আর append (উচ্চারণ অ্যাপেন্ড) শব্দের অর্থ শেষে যুক্ত করা।

```
Node *append(Node *head, int item)
{
    // প্রথমেই item দিয়ে একটি নতুন নোড তৈরি করি। এই নোডটি হবে
    // লিংকড লিস্টের শেষ নোড, তাই সে কাউকে নির্দেশ করবে না। সেজন্য
    // create_node()-এর দ্বিতীয় আর্গুমেন্টে NULL পাঠাচ্ছি।
    Node *new_node = create_node(item, NULL);

    if (head == NULL) {
        // যদি লিংকড লিস্টে কিছু না থাকে, তখন head হবে NULL। আর আমাদের
        // নতুন নোডটি হবে প্রথম নোড, তাই সেটি রিটার্ন করে দিচ্ছি। append()
        // ফাংশন যা রিটার্ন করবে, তা আমরা head-এ অ্যাসাইন করব।
        // তাই লিংকড লিস্টের head, এই নতুন নোডকে নির্দেশ করবে।
        return new_node;
    }

    // এখন লুপ চালিয়ে শেষ নোডটি খুঁজে বের করতে হবে। তাই আমরা একটি নোড
    // তৈরি করলাম, এবং শুরুতে সেখানে head-এর মান কপি করে দিলাম।
    Node *current_node = head;

    // লিংকড লিস্টের শেষ নোডের next উপাদানটি কোনো নোডকে নির্দেশ
    // করবে না, তার মান হবে NULL।
```

```
    // তাই যতক্ষণ আমরা এরকম নোড না পাচ্ছি, ততক্ষণ লুপ চলবে।
    while (current_node->next != NULL) {
        // current_node-এর পরবর্তী নোড হচ্ছে current_node->next।
        // তাই একে আমরা current_node-এ কপি করে দিচ্ছি।
        current_node = current_node->next;
    }

    // এখন current_nodeটি হচ্ছে শেষ নোড। তাই তার পরবর্তী নোড হিসেবে
    // new_node-কে অ্যাসাইন করছি।
    current_node->next = new_node;

    // head পয়েন্টার রিটার্ন করছি।
    return head;
}
```

এখন আমরা মেইন ফাংশন থেকে append() মেথড পরীক্ষা করে দেখব। আর লিংকড লিস্টের নোডগুলোর ডেটা প্রিন্ট করার জন্যও একটি ফাংশন লিখে ফেলব। তোমরা কোড দেখে দেখে টাইপ করো, তারপর কম্পাইল ও রান করে আউটপুট দেখো।

```
void print_linked_list(Node *head)
{
    Node *current_node = head;
    while (current_node != NULL) {
        printf("%d  ", current_node->data);
        current_node = current_node->next;
    }
    printf("\n");
}

int main()
{
    Node *n1, *head;

    // শুরুতে একটি নোড তৈরি করে head-কে সেই নোডে নির্দেশ করাচ্ছি। তাই
    // আমরা একটি লিংকড লিস্ট তৈরি করে ফেললাম, যেখানে কেবল একটি
    // নোড আছে। আমরা লিংকড লিস্টটি প্রিন্ট করে দেখব।
    n1 = create_node(10, NULL);
    head = n1;
    print_linked_list(head);
```

```
    // এখন আমরা লিংকড লিস্টের শুরুতে একটি ডেটা যুক্ত করছি, যার মান হচ্ছে 20।
    // তারপরে লিংকড লিস্ট আবার প্রিন্ট করে দেখছি।
    // লিংকড লিস্টটি হবে এমন : 20, 10
    head = prepend(head, 20);
    print_linked_list(head);

    // লিংকড লিস্টের শেষে একটি ডেটা (30) যুক্ত করছি।
    // লিংকড লিস্ট প্রিন্ট করলে পাব : 20, 10, 30
    head = append(head, 30);
    print_linked_list(head);

    return 0;
}
```

অনুশীলনী :

এখন তোমাদের দুটি ফাংশন তৈরি করতে হবে। প্রথম ফাংশনটি হচ্ছে count(), যার কাজ হবে লিংকড লিস্টে কয়টি নোড আছে, সেটি গণনা করা। দ্বিতীয় ফাংশনটি হচ্ছে search(), যেটি ইনপুট হিসেবে একটি লিংকড লিস্টের head পয়েন্টার ও একটি ডেটা নেবে এবং লিংকড লিস্টের যেই নোডে সেই ডেটা আছে, সেটি রিটার্ন করবে। ফাংশনগুলোর প্রটোটাইপ আমি লিখে দিচ্ছি। তোমরা যদি এতক্ষণ পর্যন্ত আমি যেসব কোড দেখিয়েছি, সেগুলো ঠিকভাবে দেখে ও বুঝে থাকো, তাহলে ফাংশনগুলো তৈরি করতে সমস্যা হবে না।

```
int count(Node *head);
Node *search(Node *head, int item);
```

ফাংশনগুলো কীভাবে তৈরি করতে হবে, বুঝতে না পারলে print_linked_list() ফাংশনটি মনোযোগ দিয়ে দেখো। search() ফাংশনের ক্ষেত্রে একটি কথা বলে দেওয়া প্রয়োজন, যেকোনো লিংকড লিস্টের যদি item একাধিকবার থাকে, তবে কেবল প্রথম যেই নোডের ডেটা item-এর সমান হবে, ফাংশনটি সেই নোডটি রিটার্ন করবে।

লিংকড লিস্ট থেকে কোনো নোড বাদ দিতে হলে কী করতে হবে, সেটি আমরা ইতিপূর্বে আলোচনা করেছি। যেই নোড বাদ দিতে হবে, সেটি যদি লিংকড লিস্টের প্রথম নোড হয়, তাহলে এরকম ব্যবস্থা, আর তা না হলে, তার আগের নোডটি আমাদের খুঁজে বের করতে হবে, এবং তারপরে কিছু

কাজ করতে হবে। আগে যেহেতু এই বিষয় নিয়ে আলোচনা করেছি, তাই সরাসরি কোডে চলে যাই–

```
/*
  remove_node() ফাংশনে প্রথম প্যারামিটার হচ্ছে লিংকড লিস্টের হেড পয়েন্টার,
  আর দ্বিতীয় প্যারামিটার হচ্ছে যেই নোডটি বাদ দিতে হবে, তার পয়েন্টার। ফাংশনটি
  থেকে আমরা লিংকড লিস্টের head রিটার্ন করব।
*/

Node *remove_node(Node *head, Node *node)
{
    if (node == head) {
        // যদি node লিংকড লিস্টের প্রথম নোড হয়, তাহলে head এখন তার
        // পরের নোডকে নির্দেশ করবে।
        head = node->next;

        // এখন আমরা node-এর জন্য বরাদ্দ মেমোরি ছেড়ে দিই।
        free(node);

        // তারপরে ফাংশন থেকে head রিটার্ন করি।
        return head;
    }

    // কোড এখানে আসার মানে হচ্ছে, node লিংকড লিস্টের প্রথম নোড নয়, বরং
    // অন্য যেকোনো নোড। তাই আমরা node-এর আগের নোডটি খুঁজে বের করব।
    // যদি সেই নোডের নাম হয় current_node,
    // তাহলে, current_node->next হবে node।
    Node *current_node = head;

    while (current_node != NULL) {
        if (current_node->next == node) {
            // current_node-এর পরের নোড হচ্ছে node।
            // অর্থাৎ, আমরা node-এর আগের নোড পেয়ে গিয়েছি,
            // তাই লুপ থেকে বের হয়ে যাব।
            break;
        }
        current_node = current_node->next;
    }
```

```
    if (current_node == NULL) {
        // current_node-এর মান NULL মানে হচ্ছে আমরা আসলে node-এর
        // আগের নোডটি খুঁজে পাইনি। লিংকড লিস্টে যদি node না থাকে, তাহলে
        // তার আগের নোড বলেও কিছু থাকবে না। তাই আমরা head যেমন ছিল,
        // তেমন রিটার্ন করে ফাংশন থেকে বের হয়ে যাব।
        return head;
    }

    // যেহেতে node মুছে ফেলতে চাইছি, তাই current_node->next এখন
    // node-এর পরবর্তী নোডকে (node->next) নির্দেশ করবে।
    current_node->next = node->next;

    // এখন আমরা node-এর জন্য বরাদ্দ মেমোরি ছেড়ে দিই।
    free(node);

    return head;
}
```

লক্ষ করো যে, ওপরের remove_node() ফাংশনটি একটি নোডকে বাদ দেয়। কিন্তু তোমাদের যদি এমন প্রয়োজন হয় যে, একটি নির্দিষ্ট ডেটা বাদ দিতে হবে, তাহলে তোমরা শুরুতে সেই ডেটা কোন নোডে আছে, সেটি খুঁজে বের করবে, তারপরে ওপরের ফাংশনটি ব্যবহার করবে। তবে এর চেয়ে ভালো বুদ্ধি হচ্ছে, ওপরের ফাংশনটি একটু পরিবর্তন করে নেওয়া, যেন দ্বিতীয় প্যারামিটারে নোড না দিয়ে ডেটা দিলেও ফাংশনটি কাজ করে। এটি কেন ভালো বুদ্ধি, সেটি তোমরা বলতে পারবে?

মেইন ফাংশন থেকে remove_node() ফাংশনটি কল করে তোমরা পরীক্ষা করে দেখতে পারো। এর জন্য আমি একটু কোড লিখে দিচ্ছি–

```
int main()
{

    Node *n1, *head, *n2;

    n1 = create_node(10, NULL);
    head = n1;
    print_linked_list(head);

    head = prepend(head, 20);
```

```
    print_linked_list(head);

    head = append(head, 30);
    print_linked_list(head);

    head = remove_node(head, n1);
    print_linked_list(head);

    n2 = head;
    head = remove_node(head, n2);
    print_linked_list(head);

    n2 = head;
    head = remove_node(head, n2);
    print_linked_list(head);

    return 0;
}
```

লিংকড লিস্টের ভেতরে কোনো জায়গায় একটি নির্দিষ্ট নোডের পরে ডেটা যুক্ত করার জন্য এখন insert() নামে একটি ফাংশন তৈরি করব।

```
/*
  ফাংশনের প্রথম প্যারামিটার হচ্ছে node, যার পরে আমরা একটি নোড যোগ করব।
  দ্বিতীয় প্যারামিটার হচ্ছে একটি ডেটা। এই ডেটা দিয়ে নোড তৈরি করে, সেই নোডটি
  node-এর পরে যোগ করতে হবে।
*/

void insert(Node *node, int item)
{
    // একটি নতুন নোড তৈরি করি, যার ডেটা হবে item এবং এই নোডটি যেহেতু
    // node ও তার পরবর্তী নোডের মাঝখানে বসবে, সেজন্য, node-এর পরবর্তী
    // নোড (node->next) হবে এই নতুন নোডের পরবর্তী নোড।
    Node *new_node = create_node(item, node->next);

    // এখন আমরা node-এর পরবর্তী নোড হিসেবে new_node-কে বসিয়ে দিই।
    node->next = new_node;
}
```

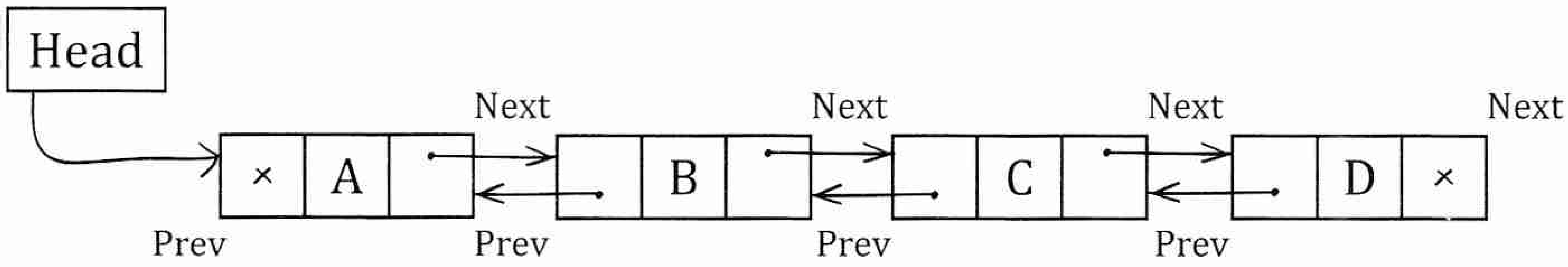
অনুশীলনী :

মেইন ফাংশনের ভেতর থেকে `insert()` ফাংশন কল করে সেটি ঠিকঠাক কাজ করে কি না, তা পরীক্ষা করে দেখতে হবে।

আমরা কম্পিউটার বিজ্ঞানের অত্যন্ত গুরুত্বপূর্ণ একটি ডেটা স্ট্রাকচার শিখে ফেললাম! এরপর আমরা আরেক ধরনের লিংকড লিস্টের সঙ্গে পরিচিত হব।

ডবলি লিংকড লিস্ট

এতক্ষণ আমরা যেই লিংকড লিস্ট আলোচনা করলাম, তার আরেকটি নাম হচ্ছে সিংগলি লিংকড লিস্ট (Singly Linked List)। আরেক ধরনের লিংকড লিস্ট আছে, যাকে বলা হয় ডবলি লিংকড লিস্ট (Doubly Linked List)। ডবলি লিংকড লিস্টের মূল বৈশিষ্ট্য হচ্ছে, এখানে প্রতিটি নোডের সঙ্গে ডেটা ও তার পরের নোডের তথ্য ছাড়াও, তার আগের নোডটির তথ্য থাকে। ছবি 6.6 দেখলেই বুঝতে পারা যাবে–



ছবি 6.6

আমরা তাহলে সিংগলি লিংকড লিস্টকে বলতে পারি একমুখী লিংকড লিস্ট আর ডবলি লিংকড লিস্টকে বলতে পারি দ্বিমুখী লিংকড লিস্ট। আমরা ইতিপূর্বে লিংকড লিস্ট বোঝার জন্য দুটি অ্যারে ব্যবহার করে একটি উদাহরণ দেখেছিলাম। সেই উদাহরণে Data ও Next নামে দুটি অ্যারে ব্যবহার করেছিলাম। ওই লিংকড লিস্টটি যদি ডবলি লিংকড লিস্ট হয়, তাহলে আমাদেরকে আরেকটি অ্যারে ব্যবহার করতে হবে, সেটির নাম আমরা দিতে পারি Prev (Previous শব্দের প্রথম চারটি অক্ষর নিয়ে)। লিংকড লিস্টটি দাঁড়াবে নিচের ছকের মতো–

Index	Data	Next	Prev
0	5	3	NULL
1			

2	12	9	6
3	8	12	0
4			
5			
6			
7	13	15	9
8			
9	3	7	2
10			
11			
12	6	2	3
13			
14			
15	9	NULL	7
16			

এই লিংকড লিস্টের প্রথম নোডটি হচ্ছে প্রথম সারির ডেটা (Index 0)। সেখানে Prev-এর ভেতরে আছে NULL, অর্থাৎ এর আগে কোনো নোড নেই। এখন লিংকড লিস্টটি আমরা শুরুর নোড থেকে শেষ নোড পর্যন্ত যেমন লিংক ধরে সবগুলো নোডে যেতে পারব, তেমনি আগের নোডের তথ্য বের করে শেষ নোড থেকে শুরুর নোড পর্যন্তও যেতে পারব। সিংগলি লিংকড লিস্টে যেমন কোনো নোড ডিলিট করতে হলে তার আগের নোডের তথ্য বের করার জন্য শুরু থেকে ওই নোড পর্যন্ত যেতে হতো, ডবলি লিংকড লিস্টে সেটি করতে হবে না। যেমন ওপরের ছবিতে যদি 9 নম্বর ইনডেক্সের নোডটি বাদ দিতে চাই, তাহলে কী করতে হবে? 9 নম্বর ইনডেক্সের নোডের আগের নোড হচ্ছে 2 (`Prev[9]`) নম্বর ইনডেক্সের নোড। আর তার পরের নোড হচ্ছে 7 (`Next[9]`) নম্বর ইনডেক্সের নোড। এখন যদি 9 নম্বর ইনডেক্সের নোড বাদ দিতে হয়, তাহলে

2 নম্বর ইনডেক্সের নোডের পরবর্তী নোড হবে 9 নম্বর ইনডেক্সের পরবর্তী নোড। তাহলে আমরা লিখতে পারি—

```
Next[9 নম্বর ইনডেক্সের আগের নোড] = 9 নম্বর ইনডেক্সের পরের নোড
বা, Next[Prev[9]] = Next[9]
```

আবার 9 নম্বর ইনডেক্সের পরের নোড, অর্থাৎ 7 নম্বর ইনডেক্সের আগের নোড হবে, 9 নম্বর ইনডেক্সের আগের নোড।

```
Prev[9 নম্বর ইনডেক্সের পরের নোড] = 9 নম্বর ইনডেক্সের আগের নোড
বা, Prev[Next[9]] = Prev[9]
```

এখন কেউ যদি নিচের ছবির প্রথম নোড থেকে শেষ নোড পর্যন্ত যায়, কিংবা শেষ নোড থেকে প্রথম নোডে আসে, সে কিন্তু আর 9 নম্বর ইনডেক্সের নোডটি খুঁজে পাবে না।

Index	Data	Next	Prev
0	5	3	NULL
1			
2	12	**7**	6
3	8	12	0
4			
5			
6			
7	13	15	**2**
8			
9	*3*	*7*	*2*
10			
11			

12	6	2	3
13			
14			
15	9	NULL	7
16			

ডবলি লিংকড লিস্টে কোনো ডেটা যোগ করার সময়ও তিনটি বিষয় খেয়াল রাখতে হবে–

1. যে নোডটি যুক্ত করতে চাইছি, তার Prev ও Next-এর মান আপডেট করতে হবে।
2. তার আগের নোডের Next-এর মান আপডেট করতে হবে।
3. তার পরের নোডের Prev-এর মান আপডেট করতে হবে।

এখন আমরা ডবলি লিংকড লিস্ট ইমপ্লিমেন্ট করব। প্রথমেই নোডের জন্য একটি স্ট্রাকচার তৈরি করি এবং সেই অনুযায়ী নতুন নোড তৈরির ফাংশনটি লিখি–

```
typedef struct node Node;

struct node {
    int data;
    Node *next;
    Node *prev;
};

Node *create_node(int item, Node *next, Node *prev)
{
    Node *new_node = (Node *) malloc(sizeof(Node));

    if (new_node == NULL) {
        printf("Error! Could Not Create A New Node\n");
        exit(1);
    }

    new_node->data = item;
    new_node->next = next;
    new_node->prev = prev;
```

```
    return new_node;
}
```

আশা করি, কোড দেখেই তোমরা বুঝতে পারছো যে, সিংগলি লিংকড লিস্টের সঙ্গে ডবলি লিংকড লিস্টের নোড স্ট্রাকচার ও নোড তৈরির ফাংশনের পার্থক্য কোথায়। এখন তাহলে prepend() ও append() ফাংশনের কোড লেখা যাক।

```
Node *prepend(Node *head, int item)
{
    Node *new_node = create_node(item, head, NULL);
    new_node->next->prev = new_node;
    return new_node;
}

Node *append(Node *head, int item)
{
    Node *new_node = create_node(item, NULL, NULL);

    if (head == NULL) {
        return new_node;
    }

    Node *current_node = head;

    while (current_node->next != NULL) {
        current_node = current_node->next;
    }

    current_node->next = new_node;
    new_node->prev = current_node;

    return head;
}
```

ওপরের কোড খুব মনোযোগ দিয়ে খেয়াল করতে হবে। সিংগলি লিংকড লিস্টের সঙ্গে এর পার্থক্যগুলো কী কী? সিংগলি লিংকড লিস্টে কোনো নোডের জন্য কেবল next-এর বিষয়টি আমাদের খেয়াল রাখতে হতো, এ ক্ষেত্রে next ও prev দুটি লিংকই আমাদের খেয়াল রাখতে হয়। যেমন— append() ফাংশনে আমরা এক জায়গায় এভাবে লিখেছি—

```
current_node->next = new_node;
new_node->prev = current_node;
```

তাই ডবলি লিংকড লিস্ট কীভাবে কাজ করছে সেটি কোড দেখে প্রয়োজনে খাতা-কলম ব্যবহার করে ছবি এঁকে বুঝে নিতে হবে। আমি চাইলে ছবিগুলো বইতে এঁকে দিতে পারতাম, কিন্তু পাঠকরা যদি নিজেরা সেই কাজটি করে, তাহলে নিঃসন্দেহে শেখাটা অনেক ভালো হবে।

এবারে আমরা নোড বাদ দেওয়ার কোড লিখব—

```
Node *remove_node(Node *head, Node *node)
{
    if (node == head) {
        head = node->next;
        head->prev = NULL;
        free(node);
        return head;
    }

    Node *previous_node = node->prev;
    Node *next_node = node->next;

    previous_node->next = next_node;
    next_node->prev = previous_node;

    free(node);

    return head;
}
```

তোমরা ওপরের ফাংশনটি দেখলে বুঝতে পারবে যে, এই ফাংশনের কমপ্লেক্সিটি হচ্ছে $O(1)$। কিন্তু সিংগলি লিংকড লিস্টের ক্ষেত্রে `remove_node()` ফাংশনের কমপ্লেক্সিটি কত ছিল, সেটি কি মনে আছে? সেটি ছিল $O(n)$। তাহলে আমরা দেখতে পাচ্ছি, ডবলি লিংকড লিস্টে নোড বাদ দেওয়া অনেক বেশি ইফিশিয়েন্ট।

অনুশীলনী :

- ডবলি লিংকড লিস্টের জন্য একটি নির্দিষ্ট নোডের পরে ডেটা যোগ করার ফাংশন লিখতে হবে সিংগলি লিংকড লিস্টে যেমন `insert()` ফাংশন ছিল।

- একটি মেইন ফাংশন তৈরি করে ডবলি লিংকড লিস্টের বিভিন্ন ফাংশন ঠিকমতো কাজ করে কি না, সেটি পরীক্ষা করতে হবে। কোনো অংশ কাজ না করলে চিন্তাভাবনা করে সেটি ঠিক করে নিতে হবে।

> 👁 কোনো কিছু ভালোভাবে আয়ত্ত করতে হলে সেই বিষয়ে চর্চা করার কোনো বিকল্প নেই। এই ওয়েবসাইটে লিংকড লিস্ট-বিষয়ক বেশ কিছু অনুশীলনী আছে :
> https://www.hackerrank.com/domains/data-structures/linked-lists
> এগুলো সমাধান করতে হবে।

লিংকড লিস্টের ব্যবহার

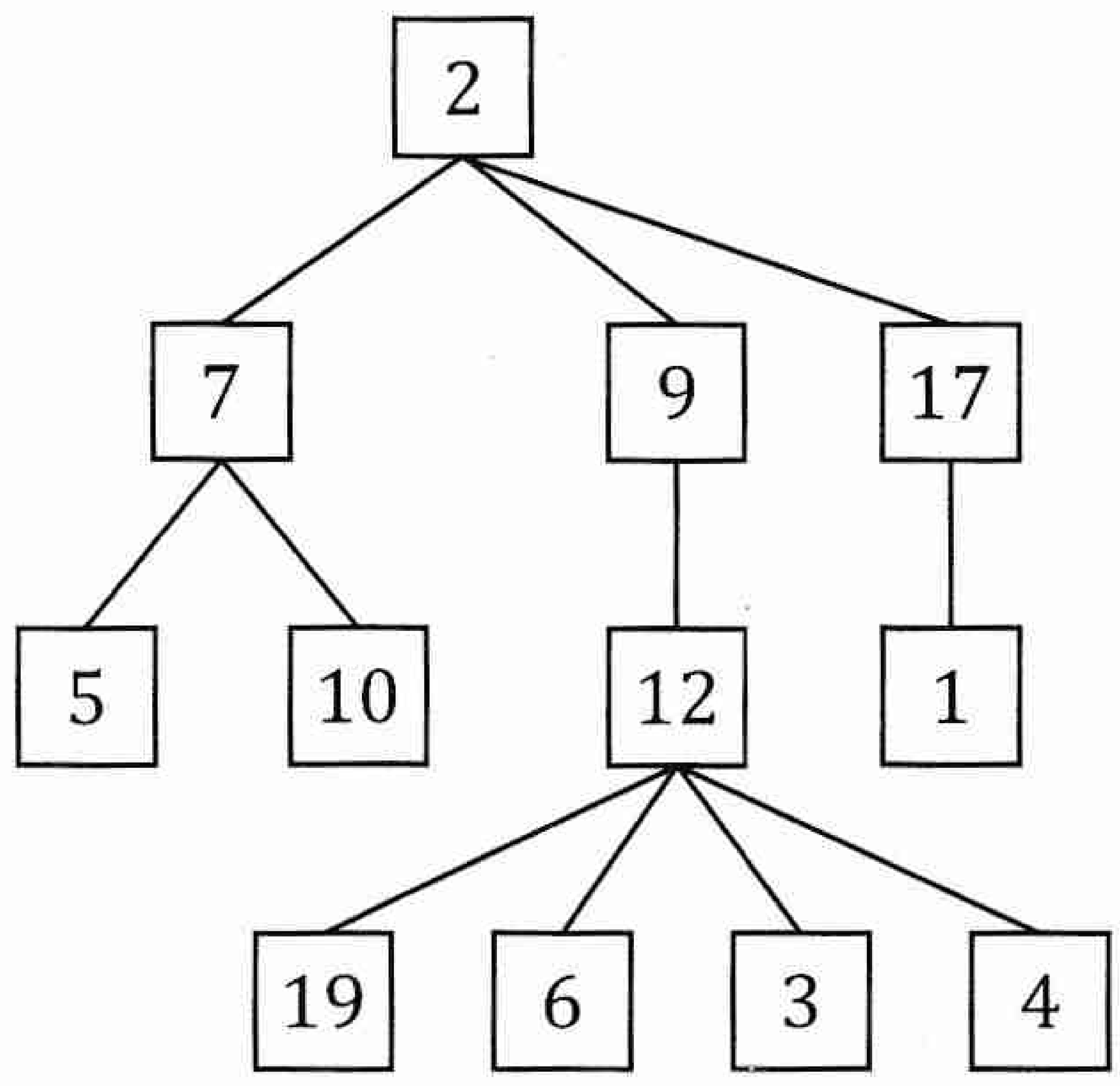
ইতিমধ্যে হয়তো অনেক পাঠকের মনেই প্রশ্ন জেগেছে যে, লিংকড লিস্ট ব্যবহার করে কোথায়? অপারেটিং সিস্টেম, বিভিন্ন প্রোগ্রামিং ল্যাঙ্গুয়েজ, ইত্যাদি তৈরির সময় বিভিন্ন জায়গায় লিংকড লিস্ট ব্যবহার করা হয়। অনেক সময় স্ট্যাক ও কিউ তৈরিতে লিংকড লিস্ট ব্যবহার করা হয়। হ্যাশ টেবিল নামে এক ধরনের ডেটা স্ট্রাকচার আছে, সেখানেও লিংকড লিস্ট ব্যবহার করা যায়। এ ছাড়া আরো অনেক ডেটা স্ট্রাকচার ও অ্যালগরিদমে লিংকড লিস্টের ব্যবহার রয়েছে।

অনুশীলনী :

- লিংকড লিস্ট ব্যবহার করে ইনসার্শন সর্ট অ্যালগরিদম ইমপ্লিমেন্ট করতে হবে এবং সেই প্রোগ্রামের কমপ্লেক্সিটি বের করতে হবে।
- একটি সিংগলি লিংকড লিস্টকে উলটে দিতে হবে। অতিরিক্ত কোনো মেমোরি ব্যবহার করা যাবে না আর প্রোগ্রামের কমপ্লেক্সিটি O(n)-এর চেয়ে বেশি হওয়া চলবে না।

অধ্যায় ৭ : ট্রি (Tree)

ট্রি (Tree) শব্দের বাংলা অর্থ গাছ। এই অধ্যায়ে আমরা জানব এমন একটি ডেটা স্ট্রাকচার সম্পর্কে, যেটি আসলে গাছের মতোই। গাছের যেমন শেকড় ও ডালপালা আছে, তেমনি এই ট্রি ডেটা স্ট্রাকচারেরও রুট (root), ব্রাঞ্চ (branch), লিফ (leaf) ইত্যাদি আছে। তবে ট্রি-এর রুট থাকে সবার ওপরে। এটি আসলে একটি নোড। তার সঙ্গে আরো বিভিন্ন নোড যুক্ত থাকে।

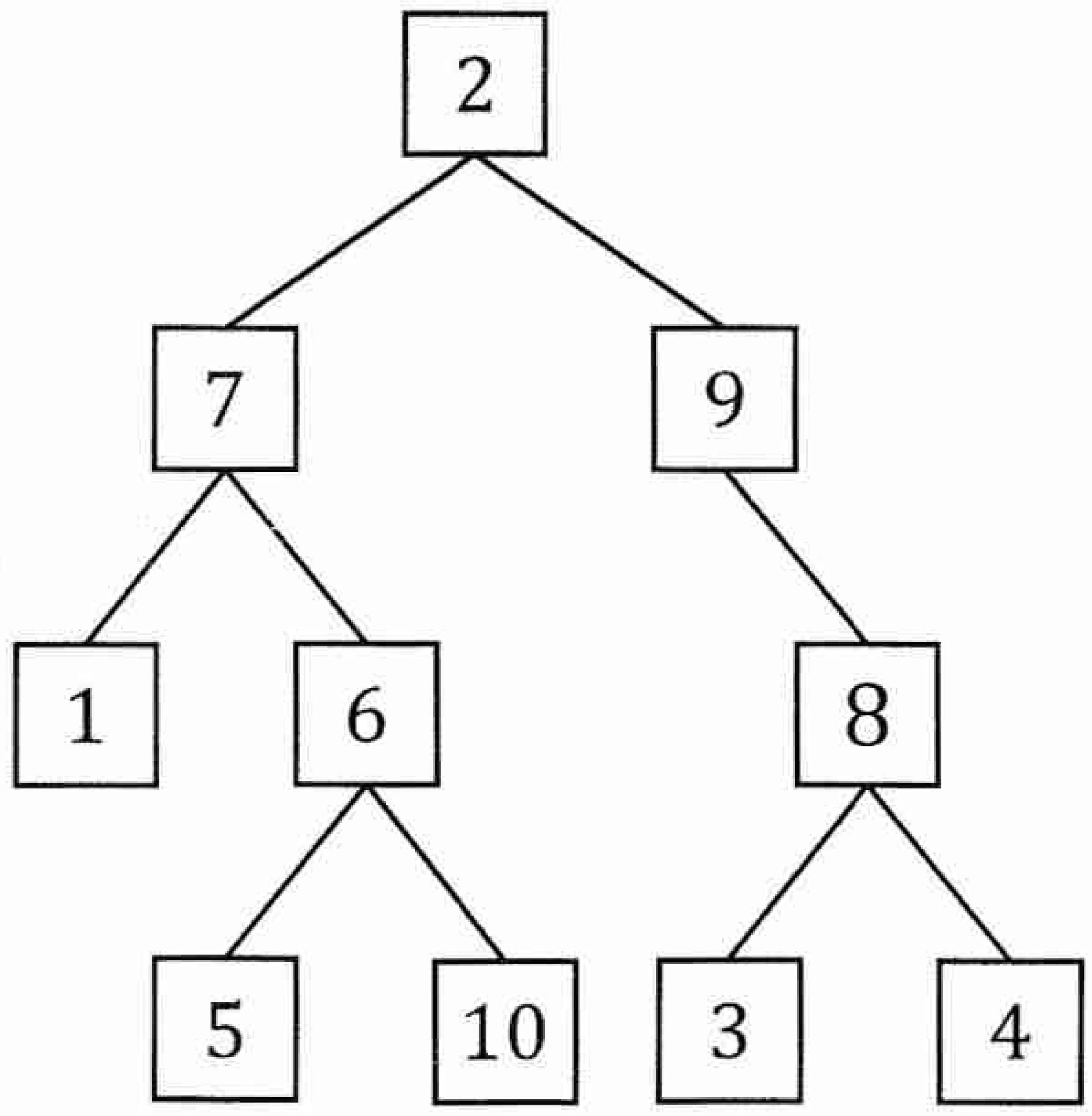


ছবি 7.1

আমরা ছবি 7.1-এ একটি ট্রি দেখতে পাচ্ছি। এই ট্রি-এর রুট নোড হচ্ছে 2। 2 থেকে তিনটি শাখা বের হয়েছে। 7, 9 ও 17 নোডগুলো সরাসরি নোড 2-এর সঙ্গে যুক্ত। তাই এদেরকে বলে নোড 2-এর চাইল্ড (child) নোড। আর নোড 2-কে বলে নোড 7 কিংবা নোড 9 কিংবা নোড 17-এর প্যারেন্ট (parent) নোড। তেমনি, নোড 7 হচ্ছে নোড 5-এর প্যারেন্ট; নোড 4 হচ্ছে নোড 12-এর চাইল্ড। আবার ওপরের ট্রি-তে 5, 10, 19, 6, 3, 4, 1-এই নোডগুলোর কোনো চাইল্ড নোড নেই। তাই এদেরকে বলা হয় লিফ (leaf) নোড।

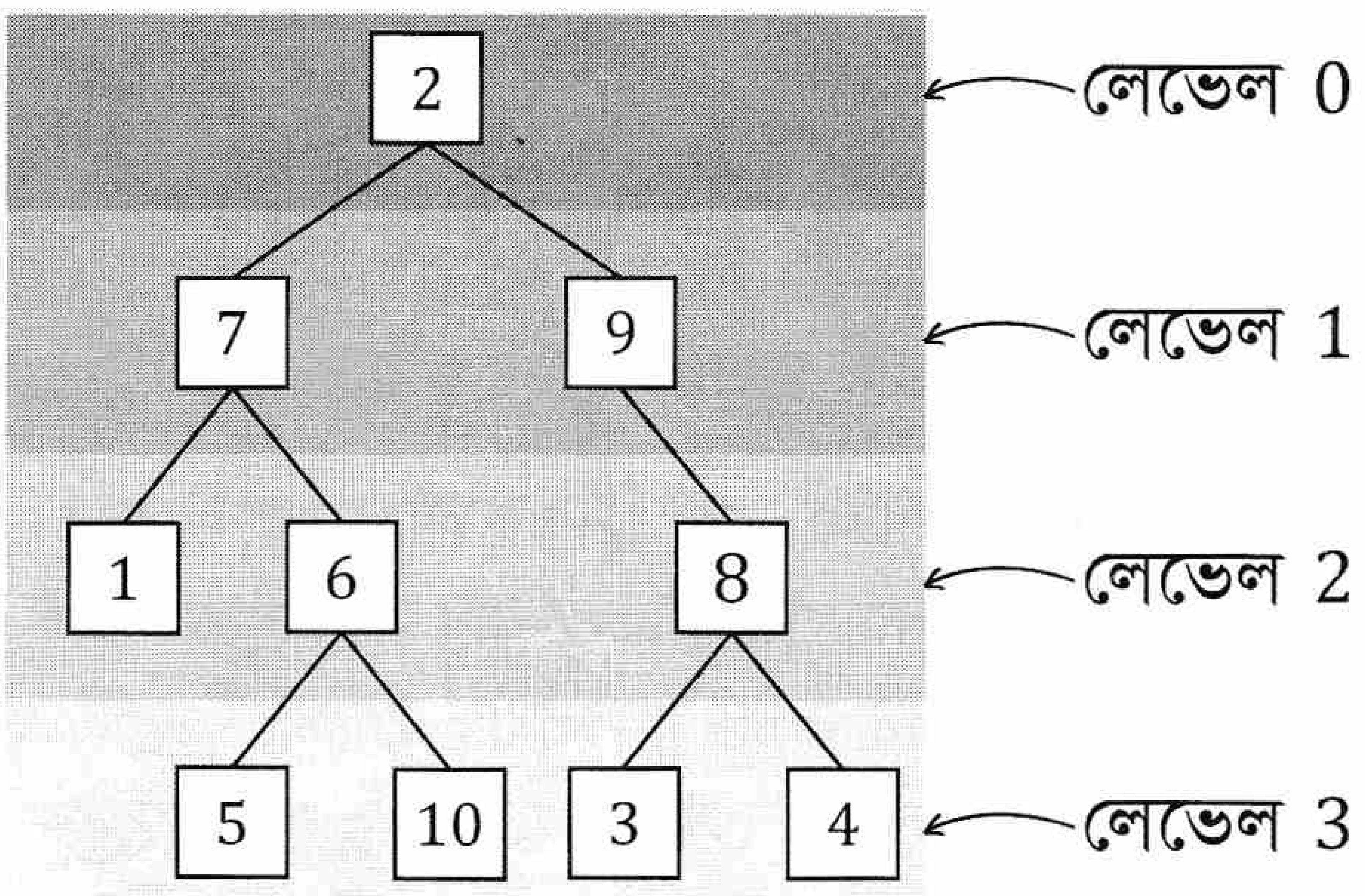
বাইনারি ট্রি

ট্রি-তে একটি নোড-এর অনেকগুলো চাইল্ড থাকতে পারে। কিন্তু যেসব ট্রি-তে একটি নোডের সর্বোচ্চ দুটি চাইল্ড থাকে, তাদেরকে বলে বাইনারি ট্রি। এই অধ্যায়ে আমরা মূলত বাইনারি ট্রি নিয়েই আলোচনা করব। আমরা ছবি 7.2-এ যে ট্রি দেখছি, এটি একটি বাইনারি ট্রি। কারণ কোনো নোডেরই দুটির বেশি চাইল্ড নোড নেই।



ছবি 7.2

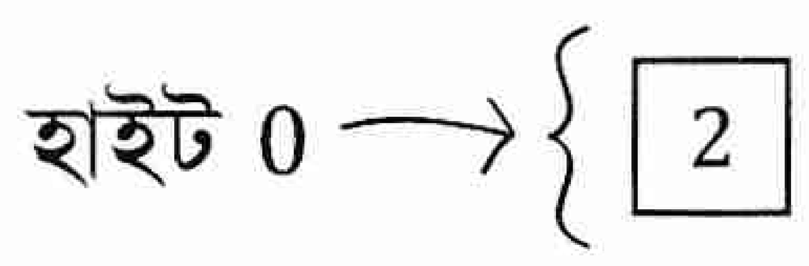
যেহেতু বাইনারি ট্রি-তে সর্বোচ্চ দুটি চাইল্ড থাকতে পারে, বাঁ দিকের চাইল্ডকে বলে লেফট (left) চাইল্ড, আর ডান দিকের চাইল্ডকে বলে রাইট (right) চাইল্ড (অনেক সময় লেফট নোড, রাইট নোডও বলা হয়)।



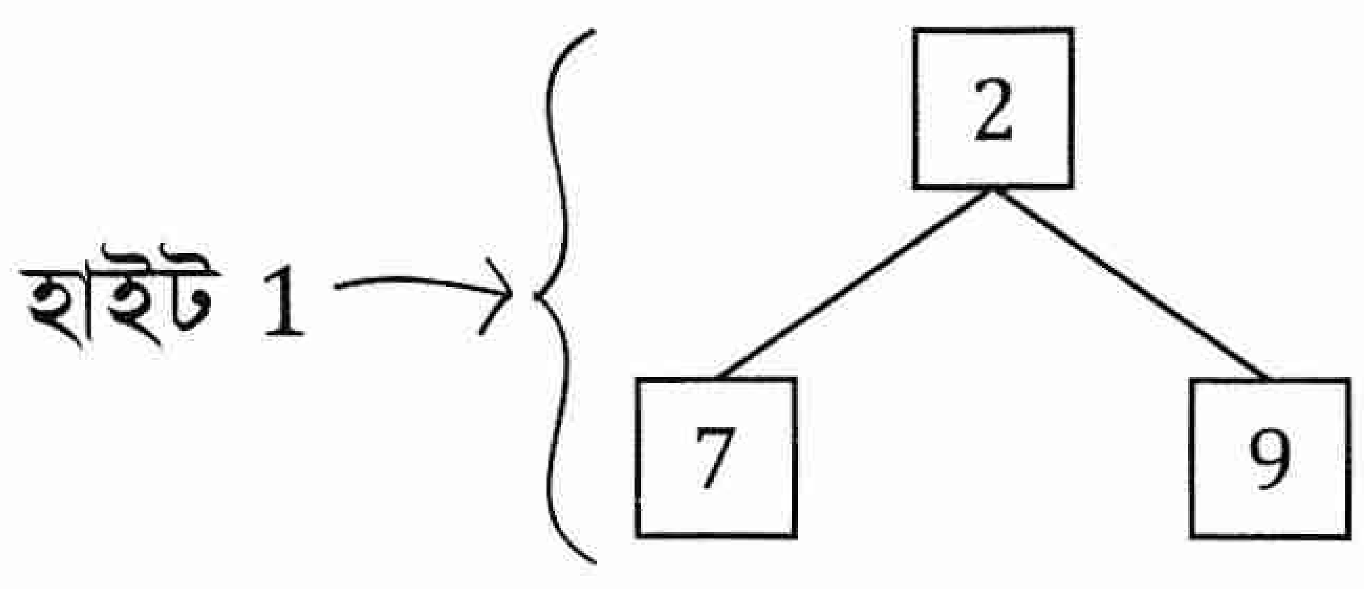
ছবি 7.3

ট্রি-এর নোডগুলোর বিভিন্ন লেভেল (level) থাকে। যেমন রুট নোডের লেভেল হচ্ছে 0। তার দুটি চাইল্ডের লেভেল হচ্ছে 1। মানে 7 ও 9 উভয় নোডেরই লেভেল হচ্ছে 1। প্রতিটি নোডের লেভেল হবে তার প্যারেন্ট নোডের লেভেলের চেয়ে এক বেশি। 1, 6 ও 8 নোডের লেভেল হচ্ছে 2। 5, 10, 3 ও 4 নোডের লেভেল হচ্ছে 3 (ছবি 7.3)।

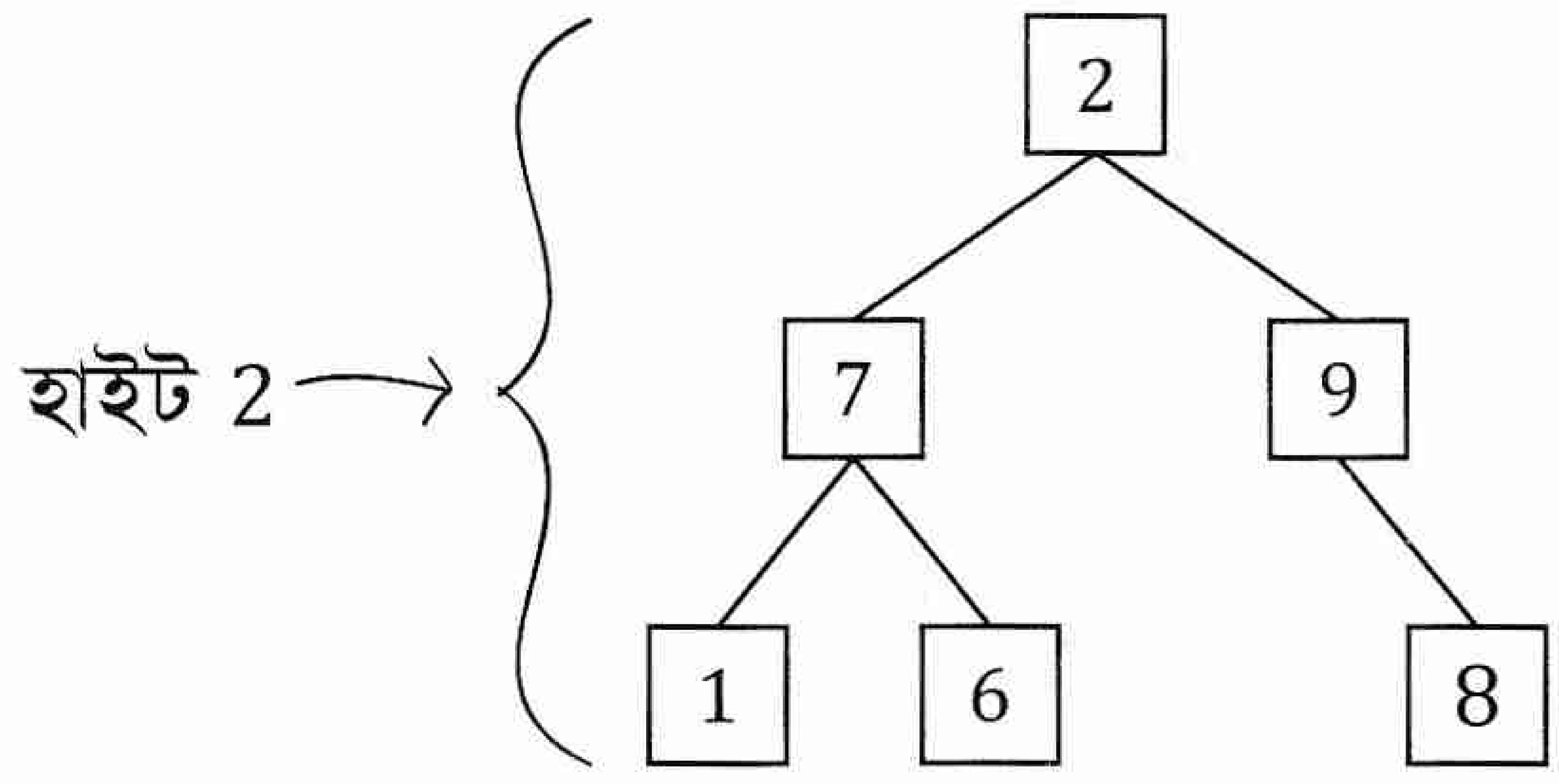
ট্রি-এর হাইট—



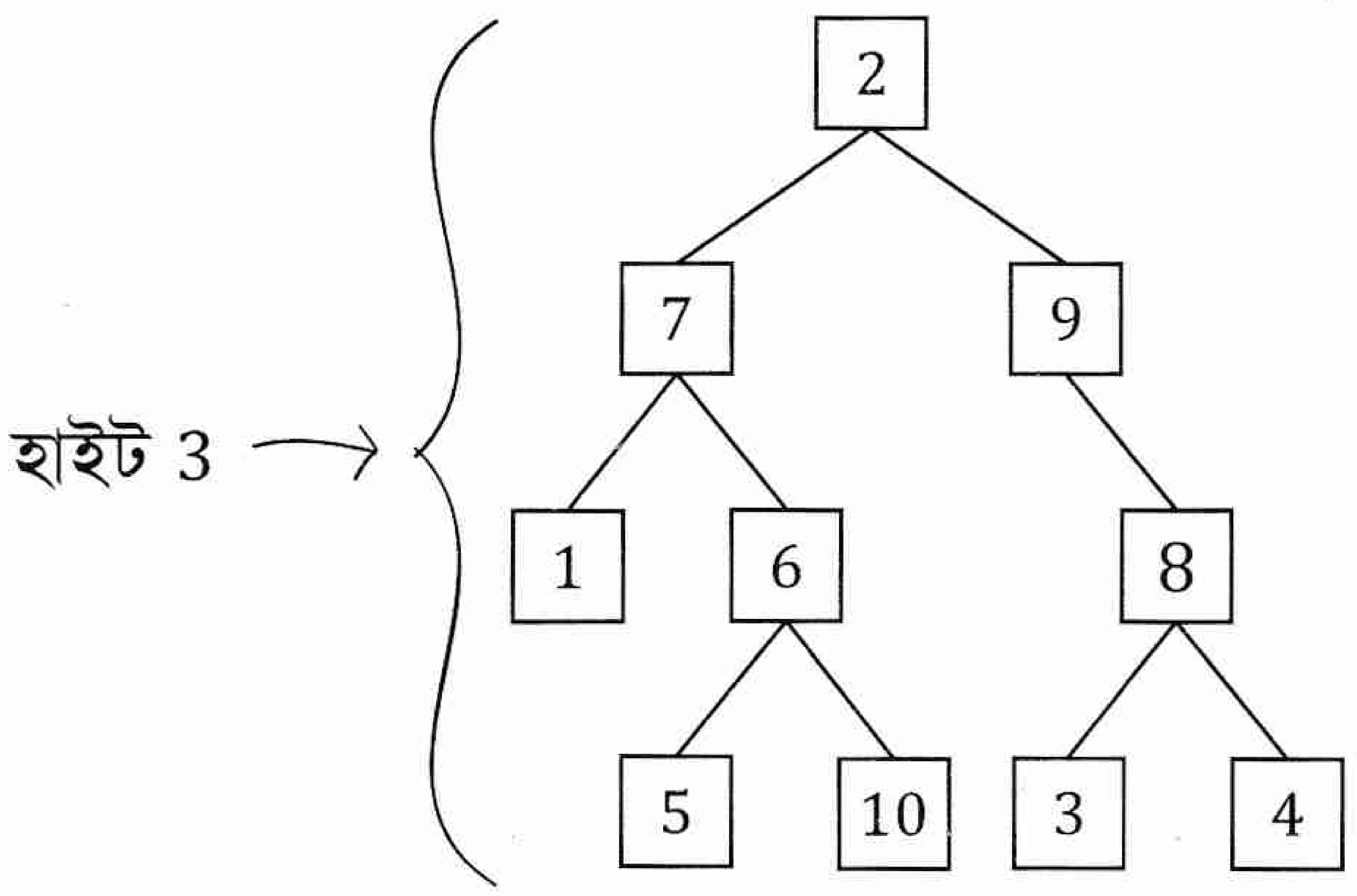
ছবি 7.4



ছবি 7.5

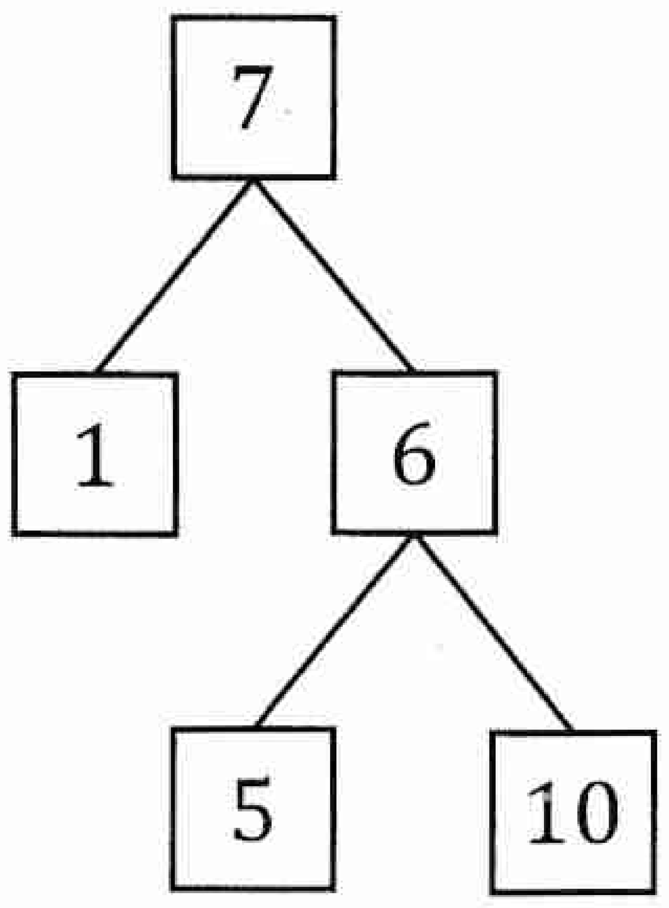


ছবি 7.6



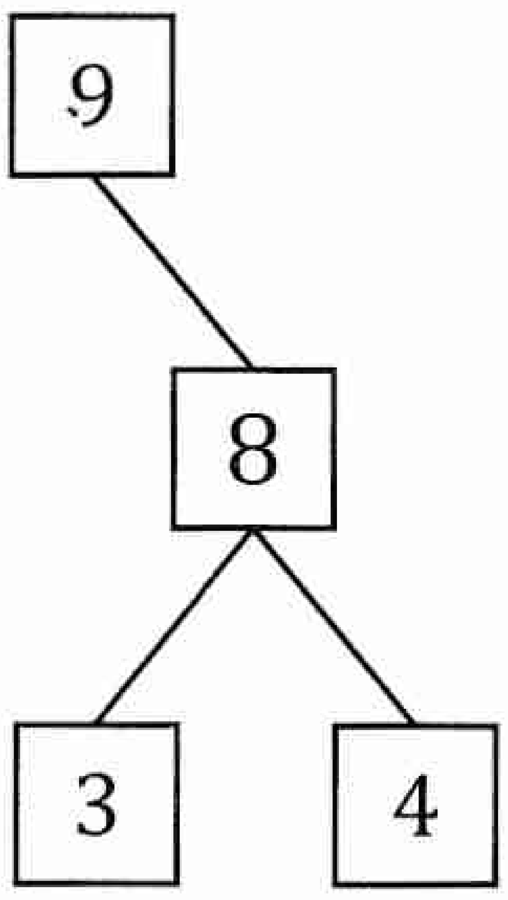
ছবি 7.7

ট্রি-এর সঙ্গে কিন্তু রিকার্শনের চমৎকার একটি সম্পর্ক আছে। যেমন, ছবি 7.7-এর রুট নোড 2-এর বাঁ দিকের নোড 7 ও তার নিচের লেভেলের নোডগুলো কিন্তু নিজেরা আরেকটি ট্রি (এটাকে মূল ট্রি-এর সাবট্রিও বলা যায়)। এই ট্রি-এর রুট নোড হচ্ছে 7 (ছবি 7.8)।



ছবি 7.8

তেমনি নোড 2-এর রাইট চাইল্ড 9 ও তার নিচের নোডগুলো মিলেও কিন্তু একটি ট্রি। এই ট্রি-এর রুট নোড হচ্ছে 9 (ছবি 7.9)।



ছবি 7.9

এভাবে প্রতিটি নোডের লেফট চাইল্ড ও রাইট চাইল্ডকে আলাদা করতে থাকলে আমরা দুটি করে ট্রি পেতে থাকব। ট্রি-এর ভেতরে ট্রি, তার ভেতরে ট্রি ... আসলে ব্যাপারটি রিকার্শনই। কারণ প্রতিটি ট্রি তার আগের ট্রি-এর চেয়ে ছোটো এবং ছোটো হতে হতে একসময় কেবল একটি নোড থাকবে, এটিই ক্ষুদ্রতম ট্রি (রিকার্শনের ক্ষেত্রে বেইজ কন্ডিশন)।

এখন একটি বাইনারি ট্রি-এর নোড-এর জন্য একটি স্ট্রাকচার তৈরি করে ফেলা যাক। এ ছাড়া আমরা নোড তৈরি করা এবং তার চাইল্ড যোগ করার জন্য ফাংশন লিখব।

```
typedef struct node Node;

struct node {
    int data;
```

```
    Node *left;
    Node *right;
};

Node *create_node(int item)
{
    Node *new_node = (Node *) malloc(sizeof(Node));

    if (new_node == NULL) {
        printf("Error! Could Not Create A New Node\n");
        exit(1);
    }

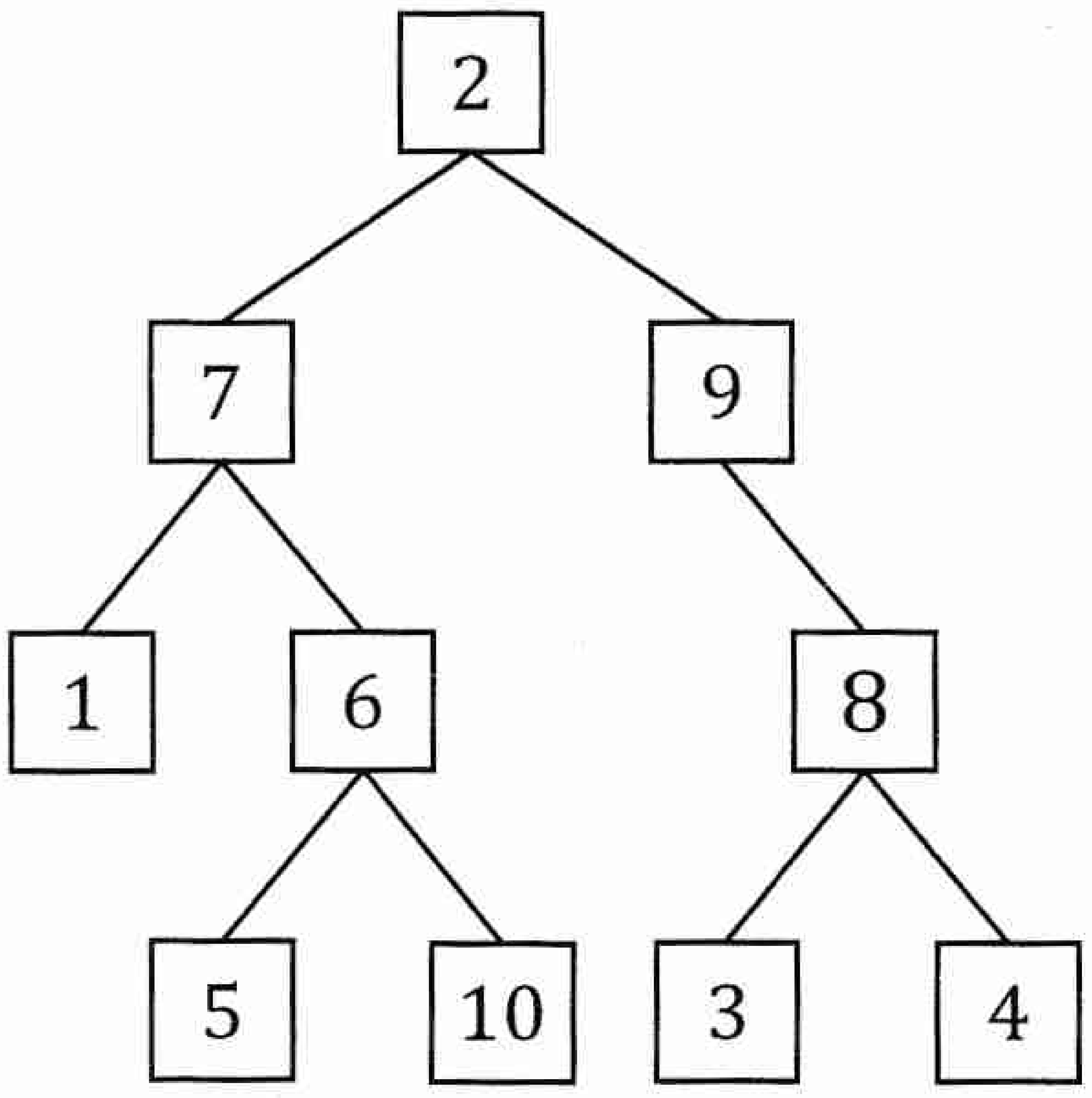
    new_node->data = item;
    new_node->left = NULL;
    new_node->right = NULL;

    return new_node;
}

void add_left_child(Node *node, Node *child)
{
    node->left = child;
}

void add_right_child(Node *node, Node *child)
{
    node->right = child;
}
```

এখন আমরা ছবি 7.10-এর ট্রি-টি তৈরি করার জন্য একটি ফাংশন লিখব। কোডটি আমাদের পরবর্তী অংশে অনেকবার ব্যবহার করতে হবে।



ছবি 7.10

```
/*
       _2_
      /   \
     7     9
    / \     \
   1   6     8
      / \   / \
     5  10 3   4
  আমরা ওপরের বাইনারি ট্রি তৈরি করে রুট নোডটি রিটার্ন করব।
*/

Node *create_tree()
{
    Node *two = create_node(2);
    Node *seven = create_node(7);
    Node *nine = create_node(9);
    add_left_child(two, seven);
    add_right_child(two, nine);

    Node *one = create_node(1);
    Node *six = create_node(6);
    add_left_child(seven, one);
    add_right_child(seven, six);

    Node *five = create_node(5);
    Node *ten = create_node(10);
    add_left_child(six, five);
```

```
    add_right_child(six, ten);

    Node *eight = create_node(8);
    add_right_child(nine, eight);

    Node *three = create_node(3);
    Node *four = create_node(4);
    add_left_child(eight, three);
    add_right_child(eight, four);

    // two নামক নোডটি হচ্ছে এই ট্রি-এর রুট।
    return two;
}

int main()
{
    Node *root = create_tree();

    printf("%d\n", root->data);

    return 0;
}
```

ট্রি ট্রাভার্স করা

একটি ডেটা স্ট্রাকচার ট্রাভার্স (traverse) করা মানে হচ্ছে, সেই ডেটা স্ট্রাকচারের বিভিন্ন উপাদান একটি নির্দিষ্ট ক্রমে অ্যাকসেস করা। যেমন, অ্যারের ভেতরের উপাদানগুলো প্রথম থেকে শেষ পর্যন্ত যাওয়া; লিংকড লিস্টের শুরুর নোড থেকে শেষ নোড, কিংবা ডবলি লিংকড লিস্টে শেষ নোড থেকে শুরুর নোড পর্যন্ত আসা। এখন ট্রি-এর ক্ষেত্রেও বিভিন্নভাবে নোডগুলো একসেস করা যায়। এখন আমরা দেখব, ট্রি ট্রাভার্স করার তিনটি অ্যালগরিদম— প্রি-অর্ডার (pre-order), পোস্ট-অর্ডার (post-order) ও ইন-অর্ডার (in-order)।

প্রি-অর্ডার ট্রাভার্সাল

প্রি-অর্ডার পদ্ধতিতে প্রথমে রুট নোড ভিজিট করা হয়। তারপরে তার বাঁ দিকের সাবট্রি প্রি-অর্ডার পদ্ধতিতে ট্রাভার্স করা হয়। তারপর তার ডান দিকের সাবট্রি প্রি-অর্ডার পদ্ধতিতে ট্রাভার্স করা হয়। তাহলে, অ্যালগরিদমটি দাঁড়াবে এরকম—

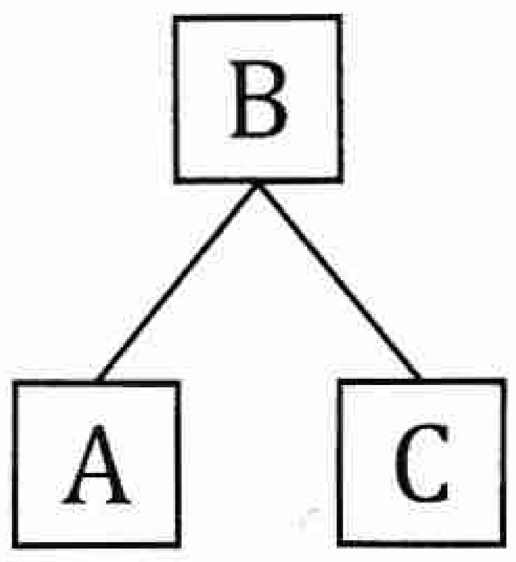
প্রি-অর্ডার (Pre-order)

ইনপুট : tree, root

ধাপ 1 : root নোড ভিজিট করতে হবে।

ধাপ 2 : বাঁ দিকের সাবট্রি ট্রাভার্স করতে হবে। অর্থাৎ, Pre-order(tree, root.left) কল করতে হবে।

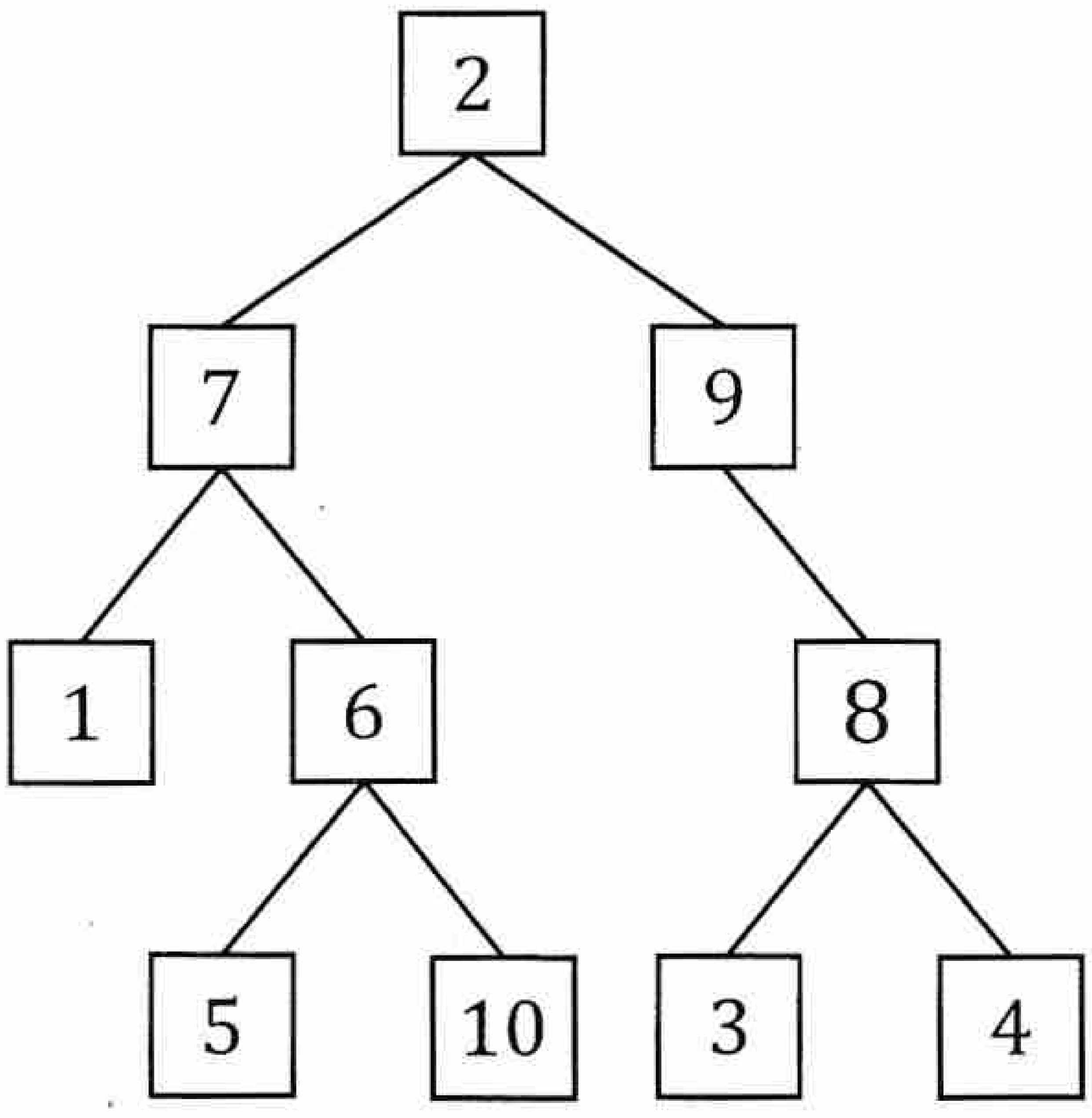
ধাপ 3 : ডান দিকের সাবট্রি ট্রাভার্স করতে হবে। অর্থাৎ, Pre-order(tree, root.right) কল করতে হবে।



ছবি 7.11

প্রি-অর্ডার পদ্ধতিতে আমরা যদি ছবি 7.11-এর ট্রি ট্রাভার্স করি, তাহলে আমরা পাব, B, A, C। কারণ প্রথমে ভিজিট করব B, তারপরে তার বাঁ দিকের সাবট্রি (অর্থাৎ যার রুট A)। এখন A-এর কোনো চাইল্ড নেই। তাই আমরা এরপর B-এর ডান দিকের সাবট্রি ট্রাভার্স করব। সেখানে C হচ্ছে একমাত্র নোড।

এখন, এই পদ্ধতিতে যদি ছবি 7.12-এর ট্রি ট্রাভার্স করি, তাহলে আমরা কোন ক্রমে নোডগুলো পাব?



ছবি 7.12

প্রথমেই পাব 2, কারণ আমরা প্রথম ধাপে রুট নোড ভিজিট করছি। তারপরে তার বাঁ দিকের সাবট্রি-এর জন্য আবার প্রিঅর্ডার ট্রাভার্স করব। তখন পাব সেই সাবট্রি এর রুট নোড, অর্থাৎ 7। তারপরে 7-এর বাঁ দিকের সাবট্রি ভিজিট করব। তখন পাব রুট নোড 1। এই নোডের আর কোনো চাইল্ড নোড নেই। তাই আমরা এখন 7-এর ডান দিকের সাবট্রি ট্রাভার্স করব। তাহলে পাব সেই সাবট্রি-এর রুট, অর্থাৎ 6। এখন 6-এর বাঁ দিকের সাবট্রি ট্রাভার্স করব, তাহলে পাব 5। 5-এর কোনো চাইল্ড নেই। তাই 6-এর ডান দিকের সাবট্রি ট্রাভার্স করব, তাহলে পাব 10। 10-এর কোনো চাইল্ড নেই। এই পর্যায়ে আমাদের নোড 7 রুটের সাবট্রি ট্রাভার্স করা শেষ। এখন নোড 2-এর বাঁ দিকের সাবট্রি ট্রাভার্স করা শেষ।

এখন ডান দিকের সাবট্রি ট্রাভার্স করব। শুরুতে পাব সেই সাবট্রি-এর রুট নোড, অর্থাৎ 9। 9-এর কোনো লেফট চাইল্ড নেই, তাই তার ডান দিকের সাবট্রিতে চলে যাব। সেই সাবট্রি-এর রুট নোড হচ্ছে 8। তারপরে 8-এর বাঁ দিকে গেলে পাব 3। 3-এর কোনো চাইল্ড নেই। তাই 8-এর ডান দিকের সাবট্রিতে চলে যাব এবং সেখানে পাব 4। 4-এর আর কোনো চাইল্ড নেই। তাহলে আমাদের ট্রি প্রিঅর্ডার পদ্ধতিতে ট্রাভার্স করা হয়ে গেল। আর আমরা যথাক্রমে এই নোডগুলো পেলাম : 2, 7, 1, 6, 5, 10, 9, 8, 3, 4।

এখন অ্যালগরিদমটি ইমপ্লিমেন্ট করে ফেলি।

```
void pre_order(Node *node)
{
    printf("%d  ", node->data);

    if (node->left != NULL) {
        pre_order(node->left);
    }

    if (node->right != NULL) {
        pre_order(node->right);
    }
}


int main()
{
    Node *root = create_tree();

    pre_order(root);
```

```
    printf("\n");

    return 0;
}
```

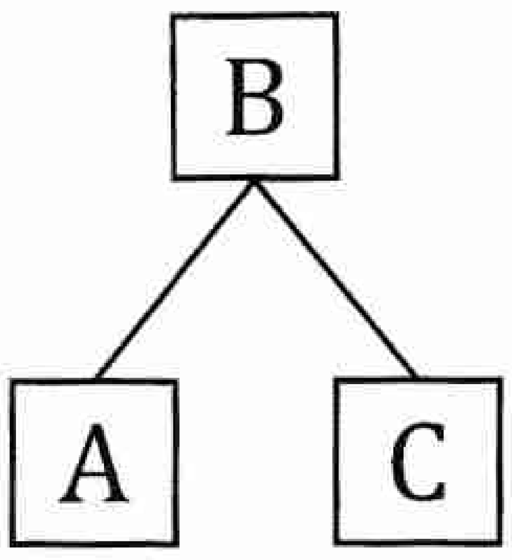
প্রোগ্রামটি রান করলে আমরা নিচের মতো আউটপুট পাব—

```
2  7  1  6  5  10  9  8  3  4
```

আমরা দেখতে পাচ্ছি যে, সঠিক ক্রমেই নোডগুলো পেয়েছি। তাহলে আমরা ঠিকঠাকভাবে প্রিঅর্ডার ট্রাভার্স করা ইমপ্লিমেন্ট করতে পেরেছি।

পোস্ট-অর্ডার ট্রাভার্সাল

পোস্ট-অর্ডার পদ্ধতিতে প্রথমে বাঁ দিকের সাবট্রি ট্রাভার্স করতে হয়, তারপরে ডান দিকের সাবট্রি ট্রাভার্স করতে হয়। তারপরে রুট নোড ভিজিট করতে হয়। তাহলে ছবি 7.13-এর ছোট্ট ট্রি পোস্ট-অর্ডার পদ্ধতিতে ট্রাভার্স করলে আমরা পাব, যথাক্রমে A, C ও B।



ছবি 7.13

এখন আমরা অ্যালগরিদমটি লিখে ফেলি—

পোস্ট-অর্ডার (Post-order)

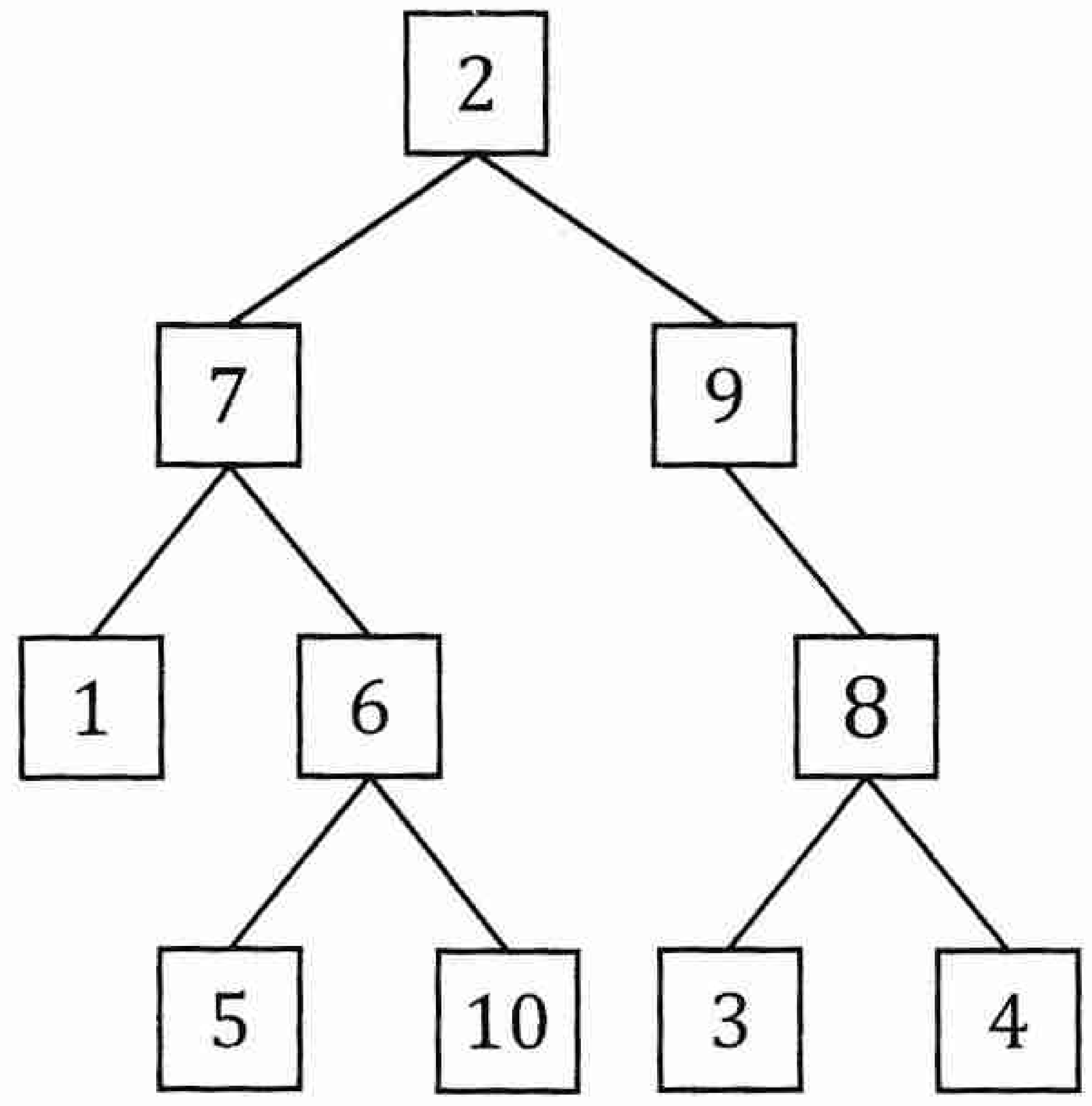
ইনপুট : tree, root

ধাপ 1 : বাঁ দিকের সাবট্রি ভিজিট করতে হবে। অর্থাৎ, Post-order(tree, root.left) কল করতে হবে।

ধাপ 2 : ডান দিকের সাবট্রি ভিজিট করতে হবে। অর্থাৎ, Post-order(tree, root.right) কল করতে হবে।

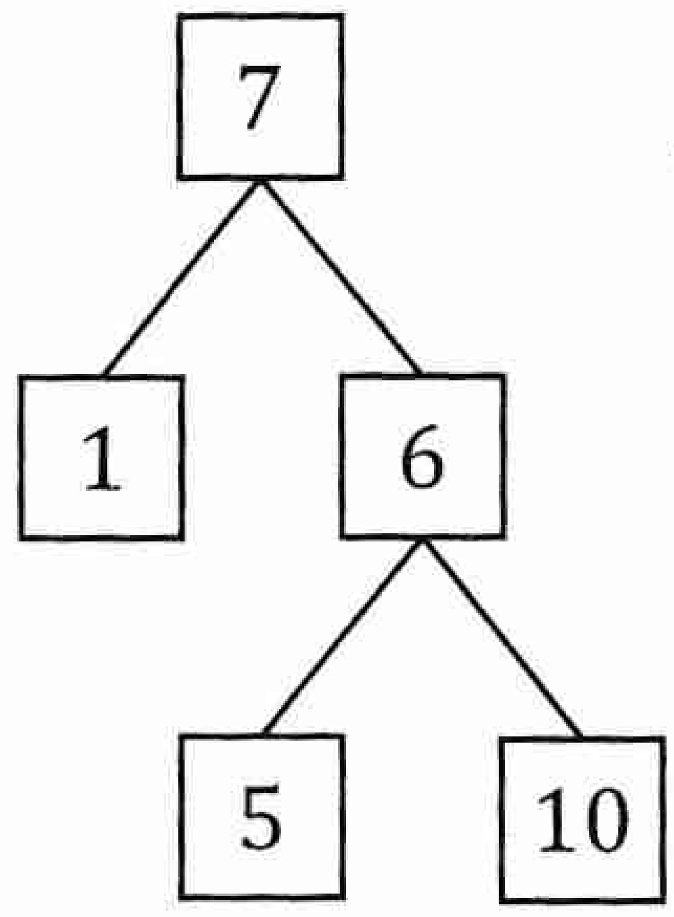
ধাপ 3 : root নোড ভিজিট করতে হবে।

এখন, এই পদ্ধতিতে ছবি 7.14-এর ট্রি ট্রাভার্স করলে, তাহলে আমরা কোন ক্রমে নোডগুলো পাব?

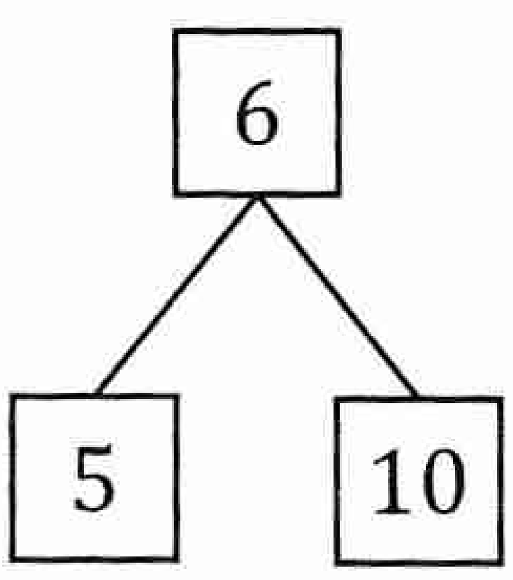


ছবি 7.14

ট্রি-এর রুট হচ্ছে 2। প্রথমে তার বাঁ দিকের সাবট্রি পোস্ট-অর্ডার অ্যালগরিদম অনুসারে ট্রাভার্স করতে হবে। সেই সাবট্রির রুট হচ্ছে 7 (ছবি 7.15)।



ছবি 7.15



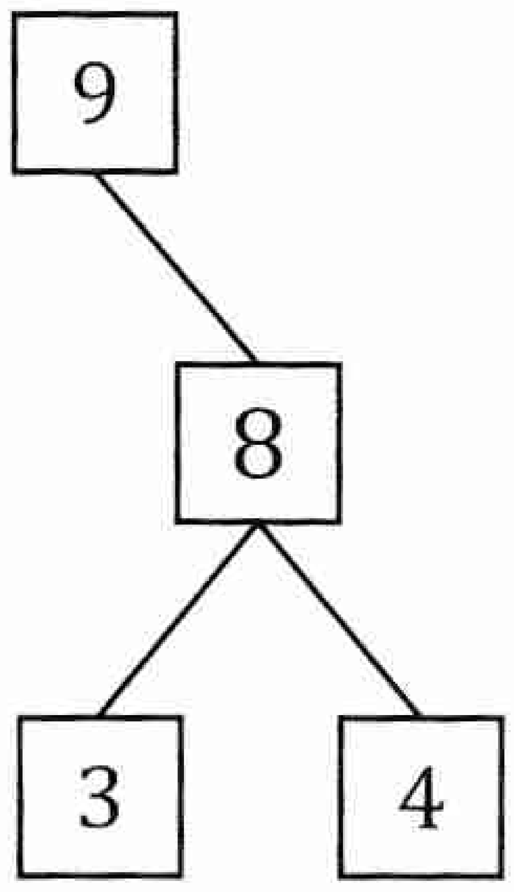
ছবি 7.16

তাই তার বাঁ দিকের সাবট্রি পোস্ট-অর্ডার পদ্ধতিতে ট্রাভার্স করতে হবে। সেই ট্রি-এর রুট হচ্ছে 1। তার বাঁ দিকের কোনো সাবট্রি নেই। তার ডান দিকেরও কোনো সাবট্রি নেই। তাই এখন আমরা সেই রুট নোডকে ভিজিট করব এবং তার মান পাব 1।

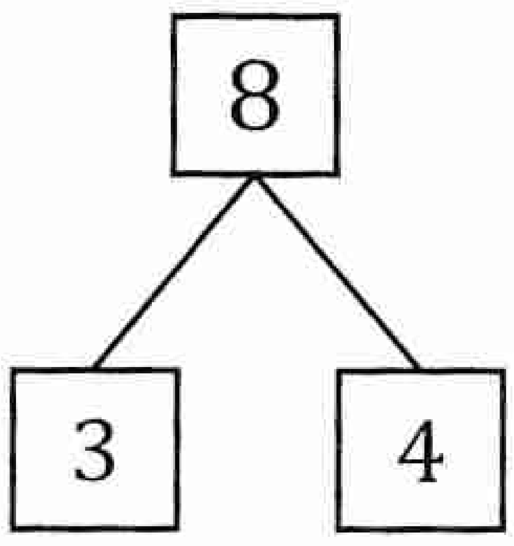
এখন 7-এর ডান দিকের সাবট্রি পোস্ট-অর্ডার পদ্ধতিতে ট্রাভার্স করব, সেই ট্রি-এর রুট নোড হচ্ছে 6 (ছবি 7.16)। ওই ট্রি-এর বাঁ দিকের সাবট্রি আবার পোস্ট-অর্ডার পদ্ধতিতে ট্রাভার্স করব। সেই ট্রি-এর রুট নোড হচ্ছে 5। এখন 5-এর বাঁ দিকে কোনো কিছু নেই। ডান দিকেও কিছু নেই। তাই আমরা এখন 5-কেই ভিজিট করব। অর্থাৎ আমরা এখন সেই নোডের মান পাব 5।

তারপরে আমরা 6-এর ডান দিকের সাবট্রি পোস্ট-অর্ডার পদ্ধতিতে ট্রাভার্স করব। সেখানে আমরা পাব 10। এখন 6-এর বাঁ দিকের সাবট্রি ও ডান দিকের সাবট্রি ট্রাভার্স করা হয়ে গেল। তাই এখন আমরা পাব 6। তাহলে এখন পর্যন্ত আমরা পেলাম 1, 5, 10, 6।

আমরা কিন্তু 7-এর বাঁ দিকের ও ডান দিকের সাবট্রি পোস্ট-অর্ডার পদ্ধতিতে ট্রাভার্স করে ফেলেছি। তাই এখন আমরা পাব বর্তমান সাবট্রি-এর রুট নোড, যার মান 7। এখন রুট নোড 2-এর বাঁ দিকের সাবট্রি আমরা পোস্ট-অর্ডার পদ্ধতিতে ট্রাভার্স করে ফেলেছি এবং আমরা পেয়েছি, যথাক্রমে 1, 5, 10, 6, 7। এবারে ডান দিকের সাবট্রি পোস্ট-অর্ডার পদ্ধতিতে ট্রাভার্স করতে হবে।



ছবি 7.17



ছবি 7.18

এই ট্রি-এর রুট নোড হচ্ছে 9 (ছবি 7.17)। তার বাঁ দিকে কিছু নেই। তাই তার ডান দিকের সাবট্রি আমরা পোস্ট-অর্ডার পদ্ধতিতে ট্রাভার্স করব (ছবি 7.18)।

তাহলে আমরা পাব, 3, 4 ও 8। তারপরে আমরা পাব 9। এখন রুট নোড 2-এর বাঁ দিকের সাবট্রি ও ডান দিকের সাবট্রি পোস্ট-অর্ডার পদ্ধতিতে ট্রাভার্স হয়ে গেল, তাই আমরা এখন পাব 2। তাহলে পুরো ট্রি আমরা পোস্ট-অর্ডার অ্যালগরিদম ব্যবহার করে ট্রাভার্স করে পেলাম 1, 5, 10, 6, 7, 3, 4, 8, 9, 2। এখন আমরা প্রোগ্রাম লিখে দেখব যে প্রোগ্রামের আউটপুটেও সংখ্যার ক্রম একই থাকে কি না।

```
void post_order(Node *node)
{
    if (node->left != NULL) {
        post_order(node->left);
    }

    if (node->right != NULL) {
        post_order(node->right);
    }

    printf("%d  ", node->data);
}

int main()
{
```

```
    Node *root = create_tree();

    post_order(root);

    printf("\n");

    return 0;
}
```

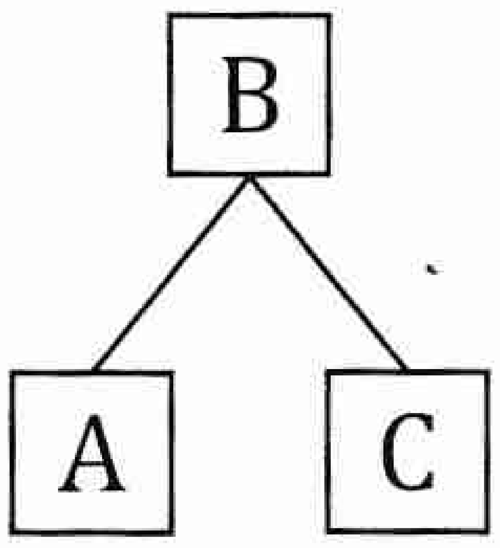
প্রোগ্রামটি রান করলে আমরা পাব—

```
1  5  10  6  7  3  4  8  9  2
```

তাহলে আমরা দেখতে পাচ্ছি, আউটপুটের সংখ্যাগুলোর ক্রম আমরা যেরকম আশা করেছিলাম, সেরকমই এসেছে।

ইন-অর্ডার ট্রাভার্সাল

ইন-অর্ডার পদ্ধতিতে প্রথমে বাঁ দিকের সাবট্রি ট্রাভার্স করতে হয়। তারপরে রুট নোড ভিজিট করতে হয়। তারপর ডান দিকের সাবট্রি ট্রাভার্স করতে হয়।



ছবি 7.19

ছবি 7.19-এর ট্রি যদি আমরা ইন-অর্ডার পদ্ধতিতে ট্রাভার্স করি, তাহলে আমরা পাব A, B, C।

তাহলে ইন-অর্ডার অ্যালগরিদমটি লিখে ফেলি—

ইন-অর্ডার (In-order)

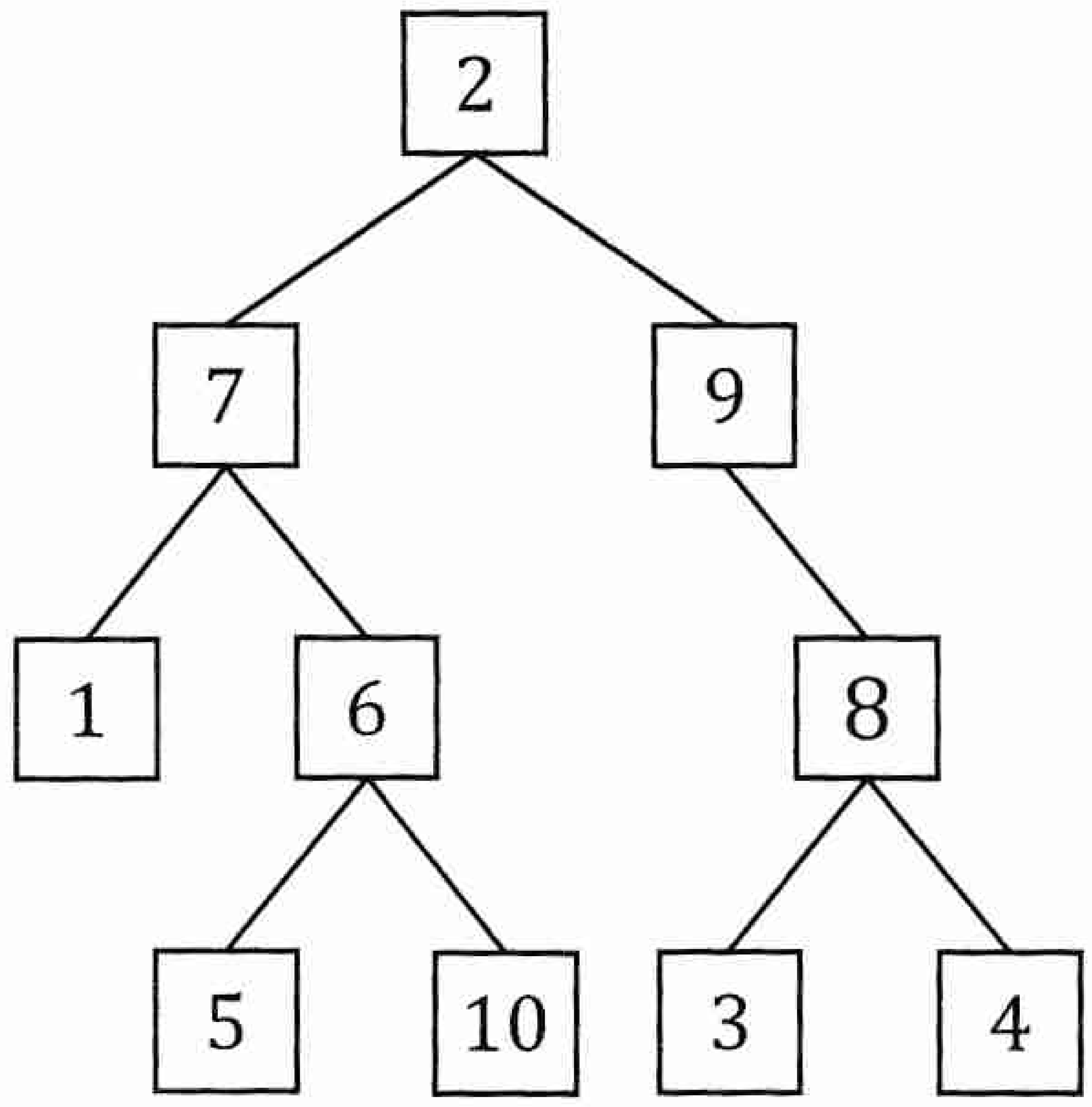
ইনপুট : tree, root

ধাপ 1 : বাঁ দিকের সাবট্রি ভিজিট করতে হবে। অর্থাৎ, In-order(tree, root.left) কল করতে হবে।

ধাপ 2 : root নোড ভিজিট করতে হবে।

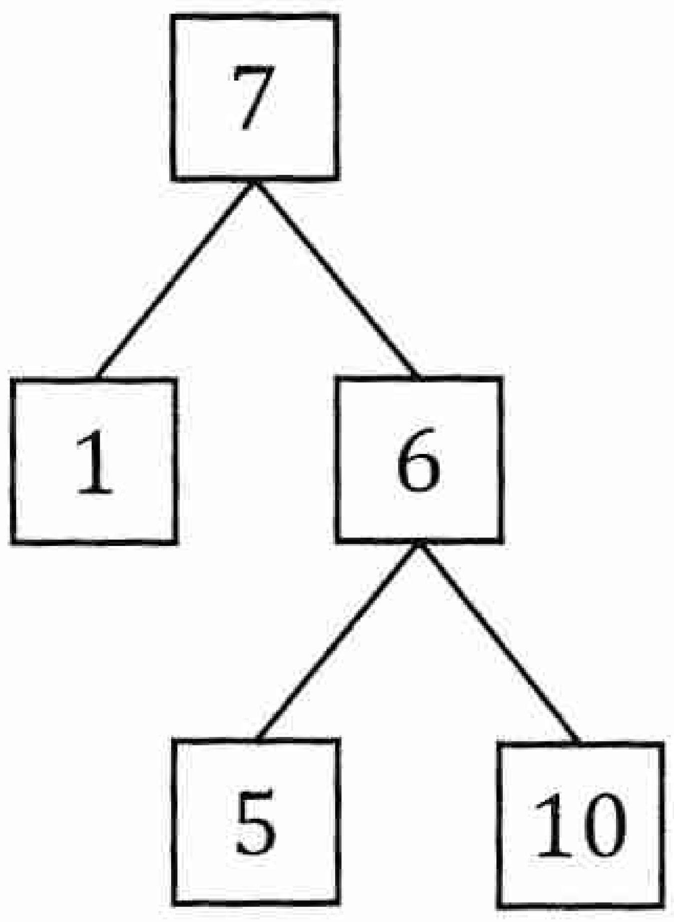
ধাপ 3 : ডান দিকের সাবট্রি ভিজিট করতে হবে। অর্থাৎ, In-order(tree, root.right) কল করতে হবে।

এখন আমরা দেখব, ছবি 7.20-এর ট্রি-টি ইন-অর্ডার অ্যালগরিদম অনুসারে ট্রাভার্স করলে কোন নোডের পর কোন নোডের মান পাওয়া যাবে।

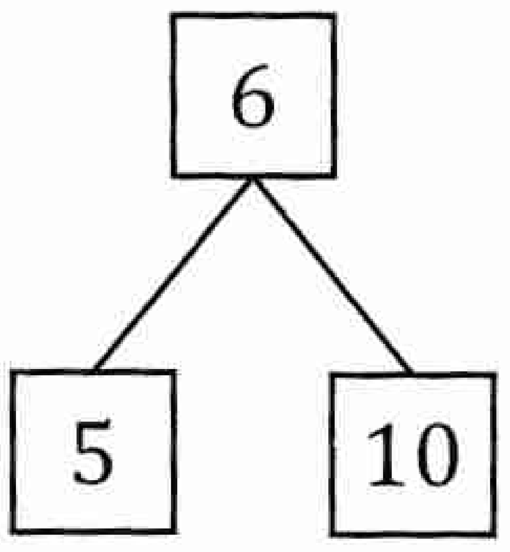


ছবি 7.20

শুরুতে রুট নোড হচ্ছে 2। তাই এর বাঁ দিকের সাবট্রি ট্রাভার্স করতে হবে (ছবি 7.21)।



ছবি 7.21

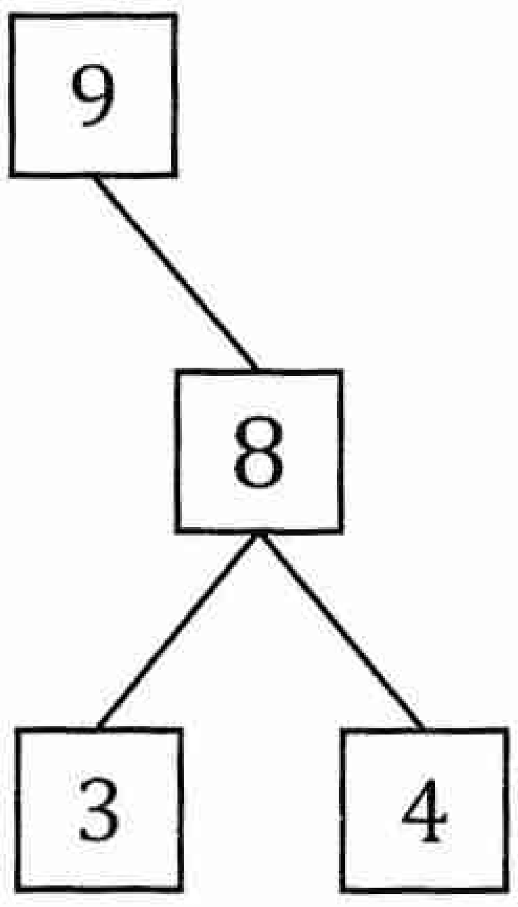


ছবি 7.22

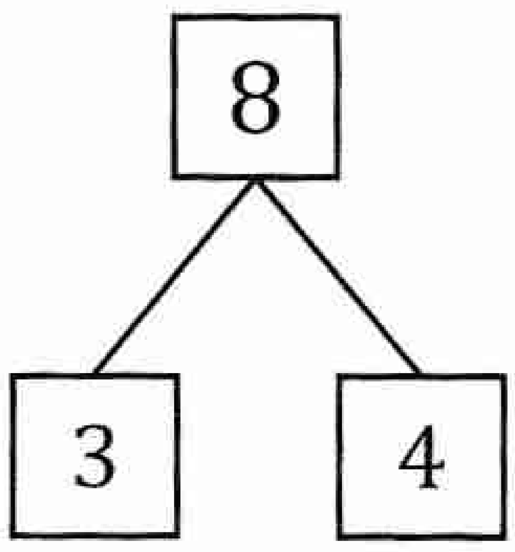
এই সাবট্রির রুট নোড হচ্ছে 7। এখন প্রথমে এর বাঁ দিকের সাবট্রি ট্রাভার্স করব। সেখানে 1-এর আর কোনো চাইল্ড নেই। তাই আমরা পাব 1। তারপরে আমরা পাব রুট নোড, অর্থাৎ 7। তারপর আমরা এর ডান দিকের সাবট্রি ট্রাভার্স করব।

সেখানে রুট নোড হচ্ছে 6 (ছবি 7.22)। তাই প্রথমে তার বাঁ দিকের সাবট্রি ট্রাভার্স করতে যাব। সেখানে একটিই নোড আছে। তাই আমরা পাব 5। তারপরে আমরা পাব রুট নোড, অর্থাৎ 6। তারপর আমরা ডান দিকের সাবট্রি ট্রাভার্স করতে যাব এবং সেখানে পাব 10, কারণ 10–এর আর কোনো চাইল্ড নেই। তাহলে এখন পর্যন্ত আমরা পেলাম 1, 7, 5, 6, 10।

এই পর্যায়ে এসে আমাদের নোড 2-এর বাঁ দিকের সাবট্রি ট্রাভার্স করার কাজ শেষ। তাই এখন আমরা রুট নোড ভিজিট করব এবং আমরা পাব 2। এখন 2-এর ডান দিকের সাবট্রি ট্রাভার্স করার পালা।



ছবি 7.23



ছবি 7.24

এখানে রুট নোড হচ্ছে 9 (ছবি 7.23)। তার বাঁ দিকে কোনো চাইল্ড নোড নেই। তাই আমরা রুট নোড 9-কে ভিজিট করব। অর্থাৎ আমরা পাব 9। এখন এর ডান দিকের সাবট্রি ট্রাভার্স করার পালা।

ইন-অর্ডার অ্যালগরিদম অনুসরণ করলে আমরা এখানে পাব, যথাক্রমে 3, 8 ও 4 (ছবি 7.24)। তাহলে পুরো ট্রি আমরা ইন-অর্ডার পদ্ধতিতে ট্রাভার্স করে পেলাম, 1, 7, 5, 6, 10, 2, 9, 3, 8, 4।

এখন আমরা ইন-অর্ডার অ্যালগরিদম ইমপ্লিমেন্ট করব। তারপরে একই ট্রি-এর জন্য প্রোগ্রাম রান করে দেখব যে, আউটপুট আমাদের বের করা ক্রমের সঙ্গে মিলে কি না।

```
void in_order(Node *node)
{
    if (node->left != NULL) {
        in_order(node->left);
    }

    printf("%d  ", node->data);

    if (node->right != NULL) {
        in_order(node->right);
    }
}

int main()
```

```
{
    Node *root = create_tree();

    in_order(root);

    printf("\n");

    return 0;
}
```

প্রোগ্রামটির আউটপুট হচ্ছে এরকম—

```
1  7  5  6  10  2  9  3  8  4
```

আমরা দেখতে পেলাম, রিকার্শন ব্যবহার করে খুব সহজেই আমরা প্রি-অর্ডার, পোস্ট-অর্ডার ও ইন-অর্ডার ট্রাভার্সাল অ্যালগরিদম ইমপ্লিমেন্ট করে ফেললাম। রিকার্শন ব্যবহার না করে স্ট্যাক ব্যবহার করেও অ্যালগরিদমগুলো ইমপ্লিমেন্ট করা যেত, তবে সে ক্ষেত্রে আমাদের কোড একটু বড়ো হতো আর কি। এই ট্রি ট্রাভার্স করার অ্যালগরিদমগুলো কম্পিউটার সায়েন্সের খুব জরুরি বিষয়, আর সে কারণে শিক্ষাপ্রতিষ্ঠানের পরীক্ষায় কিংবা চাকরির পরীক্ষায় প্রায়ই এগুলো থেকে প্রশ্ন করা হয়। এ ছাড়া এগুলো আর কী কাজে লাগে? বিভিন্ন গাণিতিক এক্সপ্রেশন চালানো, একটি ট্রি-এর ক্লোন (clone) তৈরি করা, বাইনারি সার্চ ট্রি সর্ট করা— এ ধরনের কাজ করার জন্য আমাদের ট্রি ট্রাভার্স করার অ্যালগরিদমগুলো প্রয়োজন হয়। তবে আমাদের এই মুহূর্তে এসব কাজ করার দরকার না হলেও এই পদ্ধতিগুলো বুঝতে পারলে আমরা আরো ভালো প্রোগ্রামার হিসেবে গড়ে উঠব।

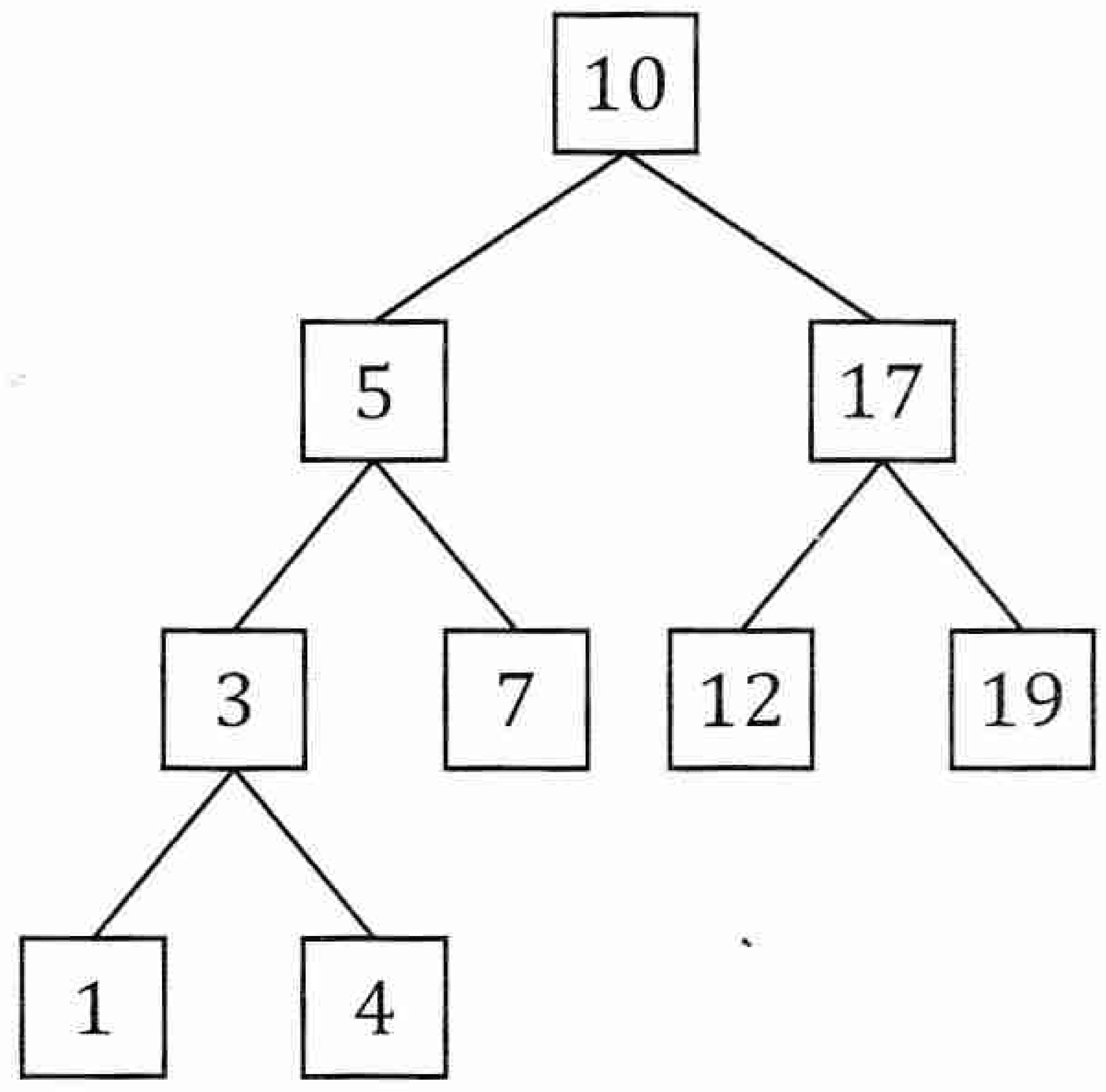
তোমরা চাইলে কয়েকটি শব্দের বাংলা অর্থ খেয়াল রাখতে পারো :

- ✓ pre মানে আগে বা পূর্বে,
- ✓ in মানে মধ্যে, আর
- ✓ post মানে পরে।

এখন দেখো pre-order, in-order, post-order ট্রাভার্স করার পদ্ধতির সঙ্গে শব্দগুলোর অর্থের কোনো মিল খুঁজে পাও কি না।

অধ্যায় ৮ : বাইনারি সার্চ ট্রি

বাইনারি সার্চ ট্রি হচ্ছে একটি বিশেষ ধরনের বাইনারি ট্রি, যেখানে প্রতিটি নোডের বাঁ দিকের চাইল্ডের মান সেই নোডের মানের চেয়ে ছোটো হয়, আর ডান দিকের চাইল্ডের মান সেই নোডের মানের চেয়ে বড়ো হয়। এই ট্রি-এর বিশেষত্ব হচ্ছে, সহজেই কোনো নোড খুঁজে বের করা যায়, অনেকটা বাইনারি সার্চের মতো। বাইনারি সার্চ ট্রি-কে সংক্ষেপে বিএসটি (BST) বলা হয়। এই অধ্যায়ে আমরা শিখব, বিএসটি থেকে কীভাবে কোনো নোড খুঁজে বের করতে হয়, কীভাবে নতুন নোড যুক্ত করতে হয়, এবং কীভাবে কোনো নোড বাদ দিতে হয়।



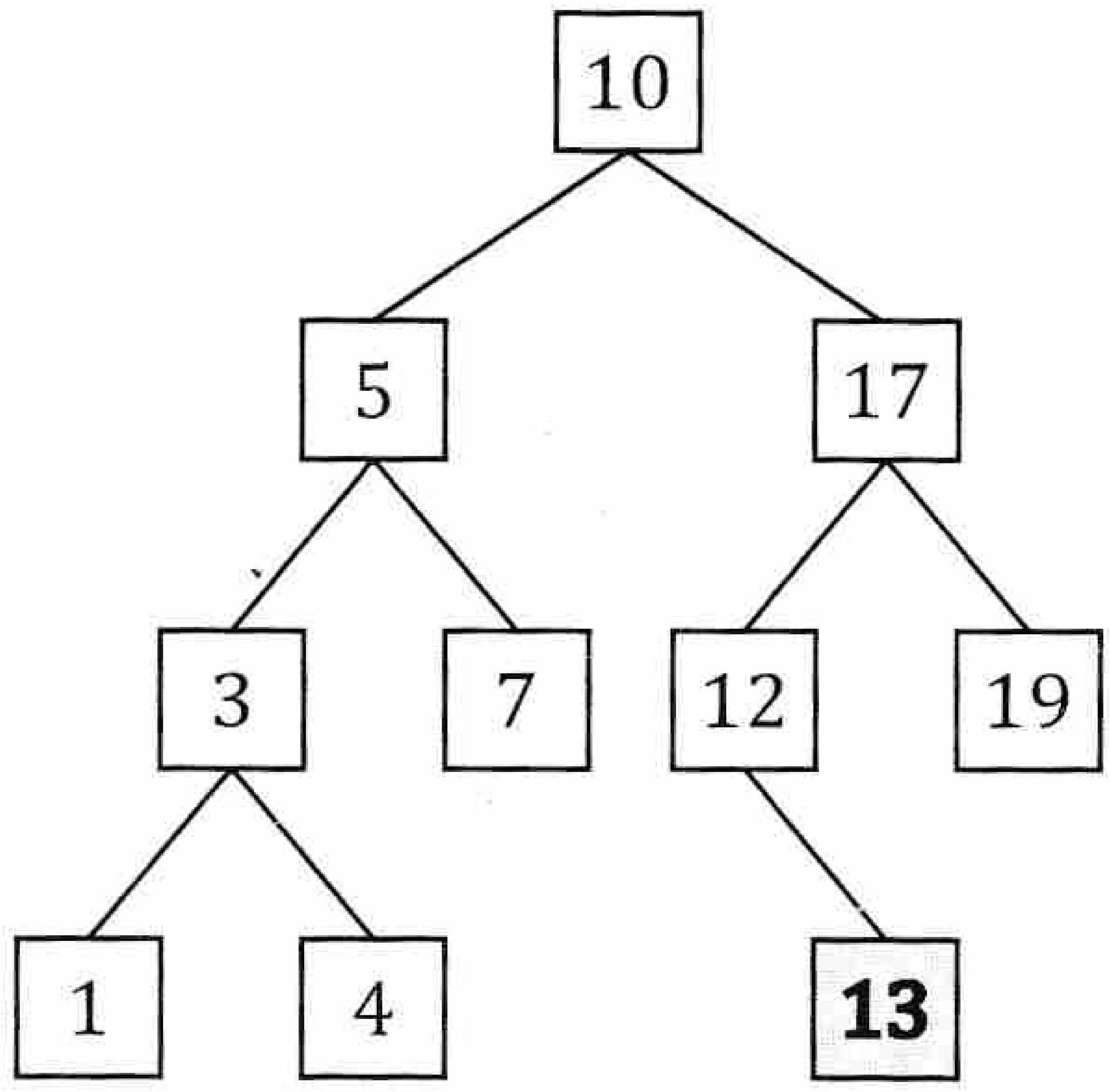
ছবি 8.1

আমরা ছবি 8.1-এ একটি বাইনারি সার্চ ট্রি দেখতে পাচ্ছি। ট্রি-এর প্রতিটি নোডের দিকে লক্ষ করলে আমরা দেখব, তার লেফট চাইল্ডের ডেটা সেই নোডের ডেটার চেয়ে ছোটো, আর রাইট চাইল্ডের ডেটা সেই নোডের ডেটার চেয়ে বড়ো।

এখন ধরা যাক, আমরা ওপরের বাইনারি সার্চ ট্রি-তে 7 নোডটি খুঁজছি। তাহলে প্রথমে আমরা রুট নোডের সঙ্গে অর্থাৎ 10-এর সঙ্গে একে তুলনা করব। 7, 10-এর চেয়ে ছোটো। তাই 7 যদি এই ট্রিতে থাকে, তাহলে সে থাকবে 10-এর বাঁ দিকের সাবট্রিতে। তারপরে আমরা 10-এর বাঁ দিকের সাবট্রি-এর রুট অর্থাৎ 5-এর সঙ্গে 7-কে তুলনা করব। 7, 5-এর চেয়ে বড়ো। তাই 7 যদি এই সাবট্রিতে থাকে, সেটি থাকবে 5-এর ডান দিকের সাবট্রিতে। এখন 5-এর ডান দিকের সাবট্রি-তে আমরা আবার 7-কে খুঁজব। সেই সাবট্রি-এর রুট নোড অর্থাৎ 7-এর সঙ্গে এখন 7-কে তুলনা করব। দুটির মান সমান, অর্থাৎ আমরা 7 পেয়ে গেলাম। আবার আমরা যদি 8 খুঁজতাম, তাহলে আমরা এই পর্যন্ত এসে দেখতাম যে, 8 হচ্ছে 7-এর চেয়ে বড়ো, তাই 7-এর ডান দিকের সাবট্রিতে থাকার কথা। কিন্তু 7-এর আর কোনো রাইট চাইল্ড নেই। তাহলে 8 এই বিএসটি-তে নেই। এই

কাজের জন্য যদি আমরা এখন একটি ফাংশন লিখি, তার কাজ হবে অনেকটা বাইনারি সার্চের মতোই। ফাংশনটি আমরা পরে লিখব।

আমরা এবারে দেখব, বিএসটি-তে কীভাবে নতুন নোড যুক্ত করতে হয়। ধরা যাক, ছবি 8.1-এর ট্রি-তে আমরা নতুন একটি নোড যোগ করতে চাই, যার ডেটা হবে 13। তাহলে প্রথমে আমরা রুট অর্থাৎ 10-এর সঙ্গে একে তুলনা করব। 13 যেহেতু 10-এর চেয়ে বড়ো, তাই এবারে 10-এর ডান দিকের সাবট্রি-এর রুট (অর্থাৎ, নোড 17)-এর সঙ্গে একে তুলনা করব। 13, 17-এর চেয়ে ছোটো। তাই আমরা এখন 17-এর বাঁ দিকের সাবট্রি-তে চলে যাব। সেই সাবট্রি-এর রুট নোড হচ্ছে 12 এবং 12-এর কোনো চাইল্ড নেই। এখন যেহেতু 13, 12-এর চেয়ে বড়ো, তাই 13-কে আমরা 12-এর রাইট চাইল্ড হিসেবে যোগ করে দেব (ছবি 8.2)।



ছবি 8.2

বাইনারি সার্চ ট্রি-তে প্রতিটি নোডের জন্য আমাদের ডেটা, বাঁ দিকের চাইল্ড ও ডান দিকের চাইল্ড ছাড়াও প্যারেন্ট নোডের তথ্য রাখা লাগবে। তাই আমরা নোডের স্ট্রাকচার ও কিছু ফাংশন একটু পরিবর্তন করি—

```
typedef struct node Node;

struct node {
    int data;
    Node *parent;
    Node *left;
    Node *right;
};

Node *create_node(int item)
{
```

```
    Node *new_node = (Node *) malloc(sizeof(Node));

    if (new_node == NULL) {
        printf("Error! Could Not Create A New Node\n");
        exit(1);
    }

    new_node->data = item;
    new_node->parent = NULL;
    new_node->left = NULL;
    new_node->right = NULL;

    return new_node;
}

void add_left_child(Node *node, Node *child)
{
    node->left = child;

    if (child != NULL) {
        child->parent = node;
    }
}

void add_right_child(Node *node, Node *child)
{
    node->right = child;

    if (child != NULL) {
        child->parent = node;
    }
}
```

এখন আমরা একটি ফাংশন লিখে ফেলি, যেটি একটি বাইনারি সার্চ ট্রি-তে নতুন নোড যুক্ত করবে এবং তারপর ট্রি-এর রুট নোডকে রিটার্ন করবে। এখানে কয়েকটি ব্যাপার মাথায় রাখতে হবে—

1. যদি ট্রি-তে আগে থেকে কোনো নোড না থাকে (অর্থাৎ বর্তমান রুট নোড NULL থাকে), তাহলে নতুন যোগ করা নোডটিই হবে ট্রি-এর রুট নোড।
2. আবার, নতুন নোডটি যদি রুট নোডের সরাসরি চাইল্ড হয়, তাহলেও রুট নোডের পরিবর্তন ঘটবে। এ কারণেই আমরা রুট নোডকে রিটার্ন করি।

```
Node *bst_insert(Node *root, Node *node)
{
    Node *parent_node, *current_node;

    if (root == NULL) {
        // root-এর মান NULL হওয়া মানে হচ্ছে, ট্রি-তে কোনো নোড নেই।
        // তাই node-কে ট্রিতে যুক্ত করলে সেটি হবে ট্রি-এর একমাত্র নোড,
        // তাই সেটিকেই আমরা রুট বানিয়ে দিচ্ছি এবং ফাংশন থেকে রিটার্ন করছি।
        root = node;
        return root;
    }

    // parent_node-এর মধ্যে আমরা node-এর প্যারেন্টকে রাখব।
    parent_node = NULL;

    // একটি লুপের মাধ্যমে আমরা ট্রি থেকে parent_nodeটি খুঁজে বের করব,
    // তারপর সেই নোডের বাঁ দিকের কিংবা ডান দিকের চাইল্ড হিসেবে node-কে
    // যুক্ত করব।
    current_node = root;

    while (current_node != NULL) {
        parent_node = current_node;

        if (node->data < current_node->data) {
            current_node = current_node->left;
        } else {
            current_node = current_node->right;
        }
    }

    if (node->data < parent_node->data) {
        // যদি node-এর ডেটা parent_node-এর ডেটার চেয়ে ছোটো হয়,
        // তাহলে node হবে লেফট চাইল্ড।
        add_left_child(parent_node, node);
    } else {
        // যদি node-এর ডেটা parent_node-এর ডেটার চেয়ে বড়ো (বা সমান)
        // হয়, তাহলে node হবে রাইট চাইল্ড।
        add_right_child(parent_node, node);
    }
```

```
    // রুট নোড (root) রিটার্ন করে দিই।
    return root;
}
```

তাহলে আমরা এই ফাংশন ব্যবহার করে একটি বিএসটি তৈরি করে ফেলি। ছবি 8.1-এ যেই ট্রি-টি দেখিয়েছি, সেটিই আমরা তৈরি করব।

```
/*
          _10_
         /    \
        5      17
       / \    /  \
      3   7 12   19
     / \
    1   4
  ফাংশনটি প্যারামিটার হিসেবে কোনো কিছু গ্রহণ করবে না।
  এবং বাইনারি সার্চ ট্রি-এর রুট নোড (Node * টাইপ) রিটার্ন করবে।
*/

Node *create_bst()
{
    Node *root, *node;

    int i;
    int ara[] = {5, 17, 3, 7, 12, 19, 1, 4};

    // প্রথমে রুট নোডটি তৈরি করি।
    root = create_node(10);

    for (i = 0; i < 8; i++) {
        // ara[i] দিয়ে একটি নোড তৈরি করি।
        node = create_node(ara[i]);

        // সেই নোডকে বাইনারি সার্চ ট্রি-তে যুক্ত করি।
        root = bst_insert(root, node);
    }

    return root;
}
```

```
int main()
{
    Node *root = create_bst();

    // বাইনারি সার্চ ট্রি প্রি-অর্ডার বা ইন-অর্ডার বা পোস্ট-অর্ডার পদ্ধতিতে প্রিন্ট
    // করে দেখতে হবে, সেই কোড লেখার দায়িত্ব তোমাদের হাতে ছেড়ে দিলাম।

    printf("\n");

    return 0;
}
```

আমরা এখন ট্রি-টি প্রিন্ট করে দেখতে চাই যে, নোডগুলো ঠিকঠাকভাবে আছে কি না। এজন্য আমরা ইন-অর্ডার, প্রি-অর্ডার কিংবা পোস্ট-অর্ডার ট্রাভার্সাল অ্যালগরিদম ব্যবহার করতে পারি। এখন মজার ব্যাপার হচ্ছে, এই তিনটি অ্যালগরিদমের মধ্যে একটি ব্যবহার করলে ট্রি-এর নোডগুলো ছোটো থেকে বড়ো ক্রমে সাজানো অবস্থায় প্রিন্ট হবে। সেই অ্যালগরিদম কোনটি? চিন্তা করে উত্তর খুঁজে বের করতে হবে। তারপরে প্রোগ্রাম লিখে মিলিয়ে দেখতে হবে।

এখন আমরা বিএসটি-তে কোনো ডেটা খুঁজে বের করার ফাংশনটি লিখে ফেলি।

```
Node *bst_search(Node *root, int item)
{
    Node *node = root;

    while (node != NULL) {

        if (node->data == item) {
            return node;
        }

        if (item < node->data) {
            node = node->left;
        } else {
            node = node->right;
        }
    }

    return node;
```

```
}
```

আমরা নিচের কোড লিখে পরীক্ষা করে দেখতে পারি—

```
int main()
{
    // প্রথমে বাইনারি সার্চ ট্রি তৈরি করে নিতে হবে।
    Node *root = create_bst();

    Node *node;

    node = bst_search(root, 7);

    if (node != NULL) {
        printf("%d\n", node->data);
    } else {
        printf("Node not found!\n");
    }

    node = bst_search(root, 8);

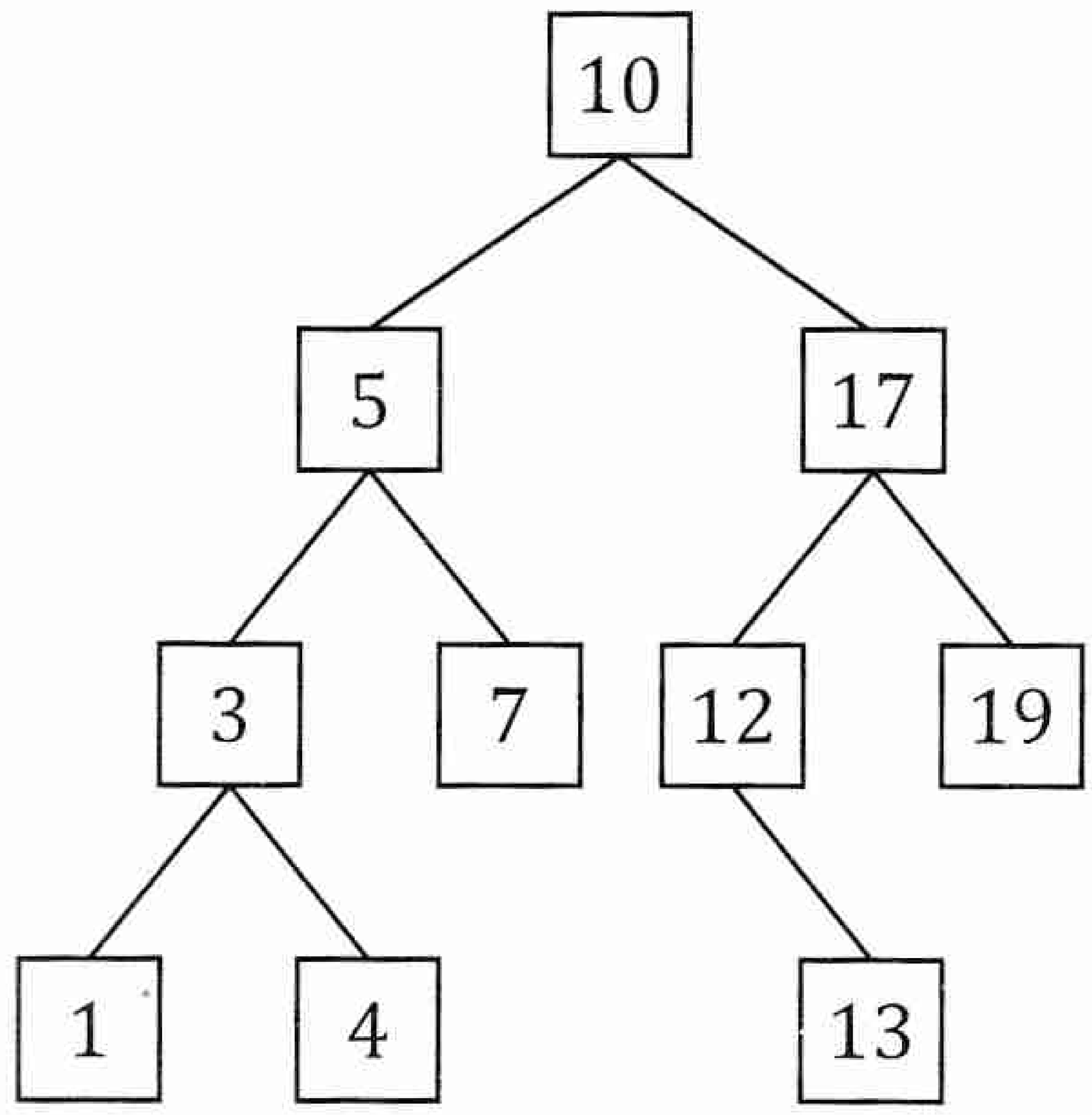
    if (node != NULL) {
        printf("%d\n", node->data);
    } else {
        printf("Node not found!\n");
    }

    return 0;
}
```

আমরা এটি রান করলে দেখব যে, 7-কে খুঁজে পাওয়া গিয়েছে, কিন্তু 8-কে খুঁজে পাওয়া যায়নি।

এখন আমাদের কাজ হচ্ছে বিএসটি থেকে কোনো নোডকে বাদ দেওয়া। এই কাজটি কোনো নোড যুক্ত করা বা খুঁজে বের করার চেয়ে একটু জটিল। কারণ এই কাজটি করতে গিয়ে বিভিন্ন রকম ব্যাপার ঘটতে পারে।

আমরা ছবি 8.3-এর ট্রি-টি লক্ষ করি—

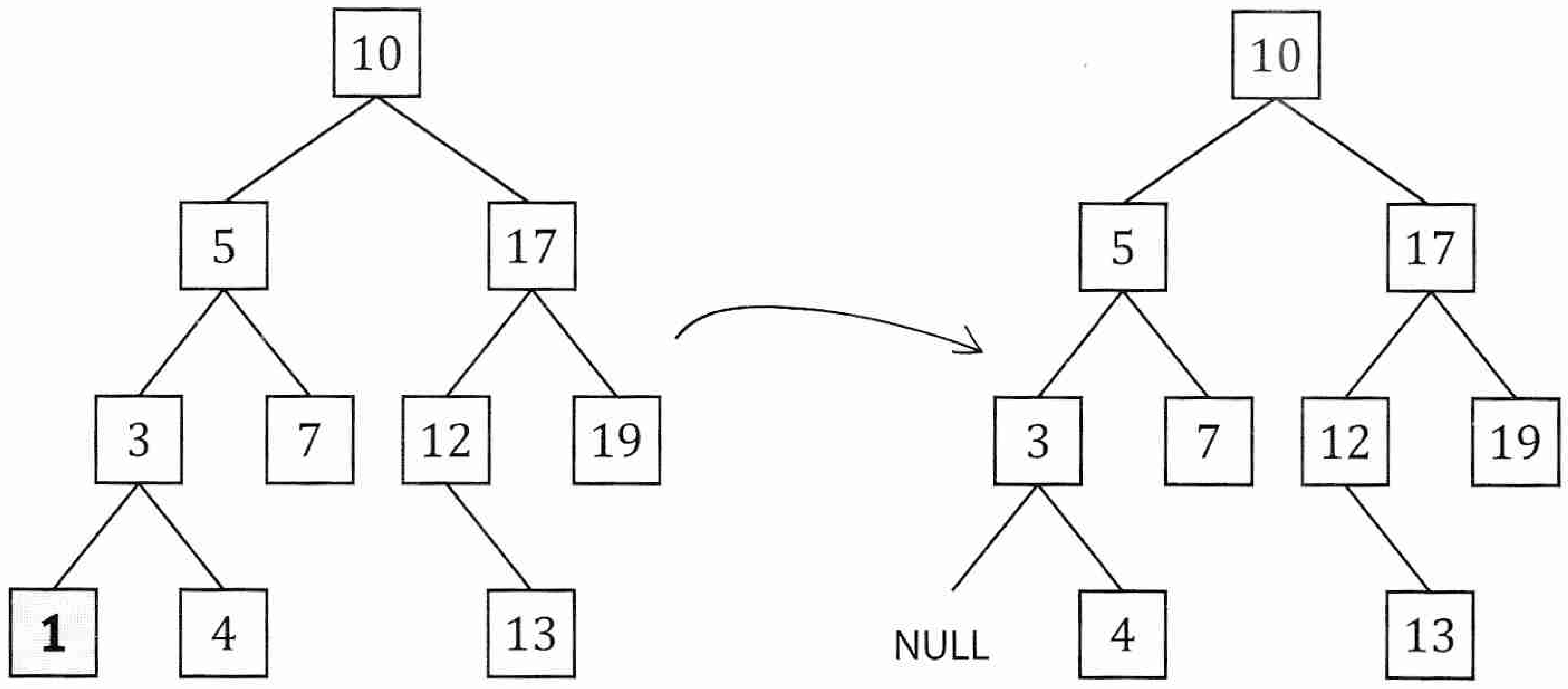


ছবি 8.3

এই বিএসটি থেকে যদি আমরা একটি নোড বাদ দিতে চাই, তাহলে সেটি হতে পারে—

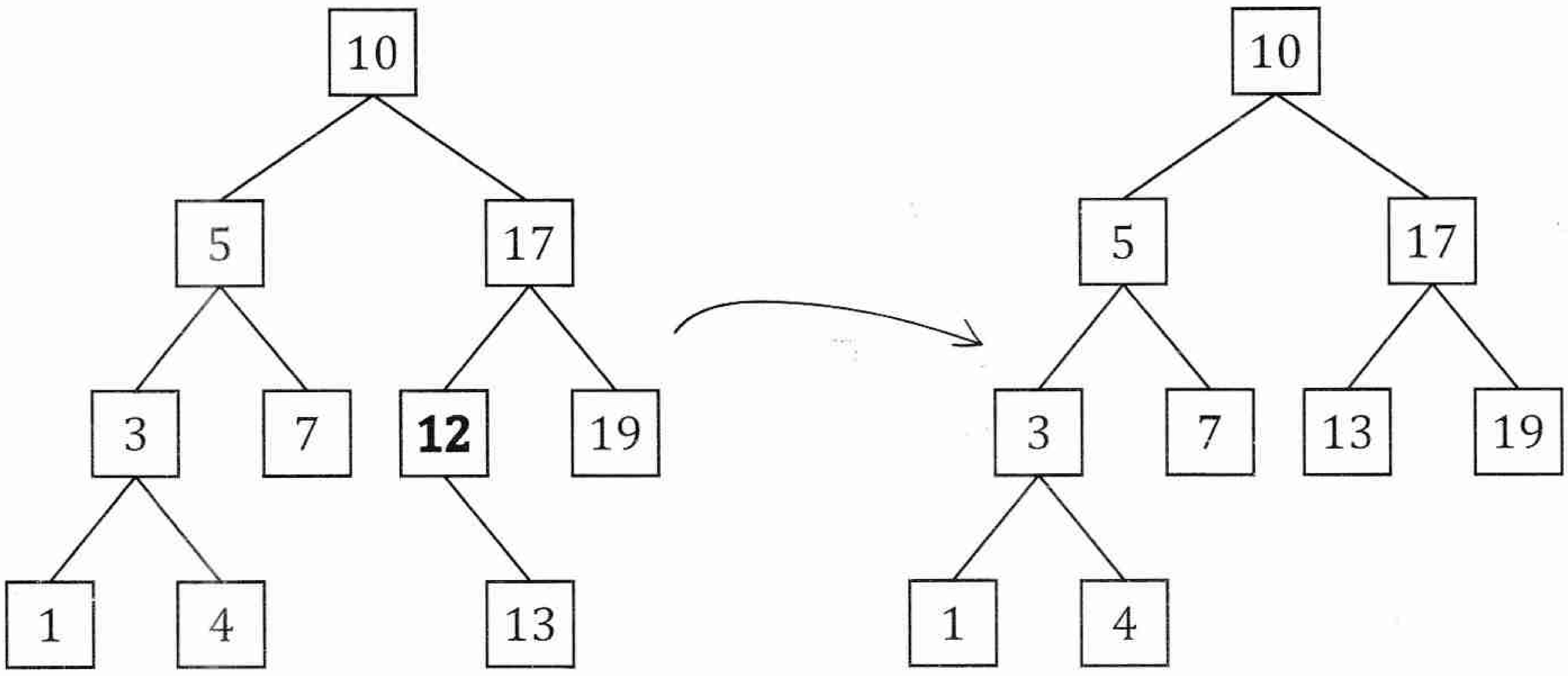
- একটি লিফ নোড, যার কোনো চাইল্ড নেই। যেমন, নোড 1, 4, 7, 13, 19।
- এমন একটি নোড, যার মাত্র একটি চাইল্ড আছে। যেমন, নোড 12।
- এমন একটি নোড, যার দুটি চাইল্ড আছে। যেমন, নোড 10, 5, 17 ও 3।

এখন প্রথম ক্ষেত্রে, আমরা সরাসরি সেই নোডটি বাদ দিয়ে দেব। তাহলে আমাদেরকে ওই নোডের প্যারেন্ট নোডের তথ্য হালনাগাদ (আপডেট) করতে হবে। যেমন ওপরের ট্রি থেকে আমি যদি 1 নোডটি বাদ দিই, তাহলে তার প্যারেন্ট, অর্থাৎ, 3-এর লেফট চাইল্ডকে NULL করে দিতে হবে।



ছবি 8.4: লিফ নোড 1 বাদ দিয়ে দেওয়া হলো

দ্বিতীয় ক্ষেত্রে, আমরা একমাত্র চাইল্ডকে তার প্যারেন্টের জায়গায় বসিয়ে দেব। যেমন আমরা যদি 12 নোডটি বাদ দিতে চাই, তাহলে তার একমাত্র চাইল্ড 13-কে 12-এর জায়গায় নিয়ে আসব। তাহলেই হয়ে গেলো।



ছবি 8.5: নোড 12 বাদ দিয়ে তার জায়গার নোড 13 বসিয়ে দেওয়া হলো

এখন আসা যাক তৃতীয় বিষয়ে, যেখানে আমরা যেই নোডটি বাদ দিতে চাই, সেই নোডের দুটি চাইল্ড আছে। ওপরের ছবিতে এরকম নোড হচ্ছে 10, 5, 17, 3। তো আমরা নোড বাদ দেওয়া বা মুছে ফেলার এই কাজটি করার আগে বাইনারি সার্চ ট্রি সম্পর্কে আরেকটু গবেষণা করব।

কোনো বিএসটি থেকে সবচেয়ে ছোটো নোড ও সবচেয়ে বড়ো নোড বের করার উপায় কী? বাইনারি সার্চ ট্রি-এর বৈশিষ্ট্য অনুযায়ী, সবচেয়ে ছোটো নোড হবে ট্রি-এর সবচেয়ে বাঁ দিকের নোড। অর্থাৎ আমরা ট্রি-এর রুট থেকে তার বাঁ দিকের চাইল্ড নোডে যাব। তারপর তার বাঁ দিকের চাইল্ড নোডে যাব। এভাবে যতক্ষণ বাঁ দিকের নোডের লেফট চাইল্ড থাকবে, ততক্ষণ পর্যন্ত বাঁ দিকের চাইল্ড নোডে যেতে থাকব। তাহলেই আমরা ট্রি-এর সবচেয়ে বাঁ দিকের নোডে চলে যাব এবং ছোটো নোডটি পেয়ে যাব। আর একইভাবে রুট থেকে তার ডান দিকের চাইল্ড নোডে যেতে থাকলে আমরা সবচেয়ে বড়ো নোডটি পেয়ে যাব। এখন আমি বিএসটি-এর সবচেয়ে ছোটো নোড বের করার জন্য একটি ফাংশন লিখব, যেটি আমরা পরবর্তী সময়ে ব্যবহার করব।

```
Node *bst_minimum(Node *root)
{
    Node *node = root;

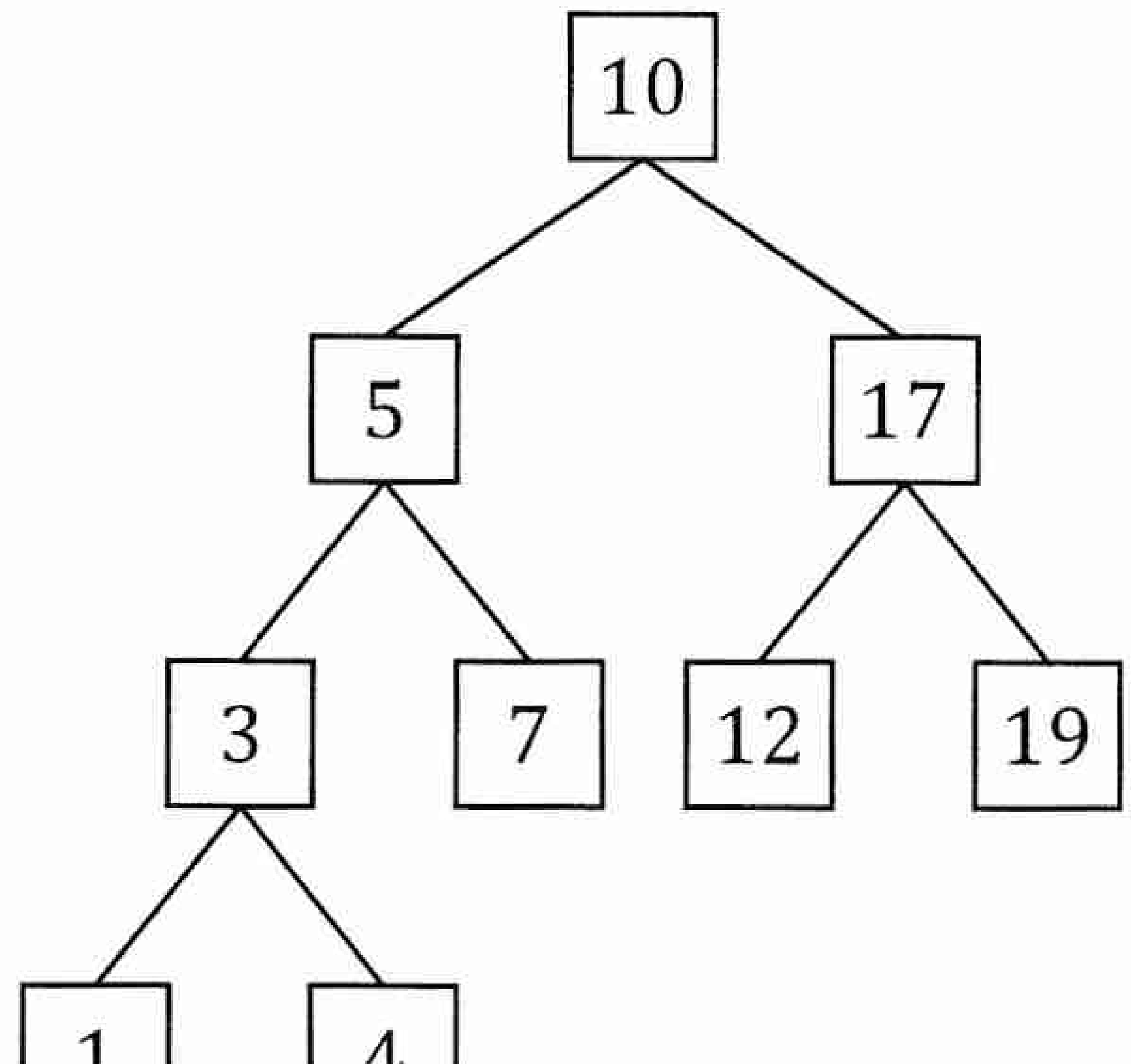
    while (node->left != NULL) {
        node = node->left;
    }
```

```
    return node;
}
```

এখন আমি যদি নোডগুলো ছোটো থেকে বড়ো ক্রমে সাজানো অবস্থায় পেতে চাই তাহলে কী করব? সবচেয়ে বাঁ দিকের নোডে পৌঁছলে তো আমরা সবচেয়ে ছোটো নোডটি পেয়ে যাব। আগের ছবি অনুসারে সেটি 1। এর পরের সবচেয়ে ছোটো নোডটি হচ্ছে ওই নোডের প্যারেন্ট। ছবিতে সেটি হচ্ছে 3। 3-এর পরের সবচেয়ে ছোটো নোডটি (মানে 3-এর চেয়ে বড়ো, কিন্তু বাকি সব নোডের চেয়ে ছোটো) পাওয়া যাবে তার ডান দিকের সাবট্রি-তে। সেই সাবট্রি-এর সবচেয়ে বাঁ দিকের নোডটি হচ্ছে 3-এর পরবর্তী নোড। আমাদের ছবিতে সেটি হচ্ছে 4। 3-এর ডান দিকের সাবট্রি এর সবগুলো নোড ভিজিট করা হয়ে গেলে তারপর আমরা তার প্যারেন্ট অর্থাৎ-5-কে ভিজিট করব। তাহলে ছবির ট্রি তে আমরা পাব-5। তারপরের নোডটি হচ্ছে 7, তারপর 10। 10-এর পরবর্তী নোড কোনটি হবে? তার ডান দিকের যেই সাবট্রি আছে, তার সবচেয়ে বাঁ দিকের নোড, অর্থাৎ 12। এভাবে আমরা একে একে পেয়ে যাব যথাক্রমে 13, 17, 19। আসলে আমরা যেই কাজটি করলাম, সেটি হচ্ছে, ট্রি-কে ইন-অর্ডার পদ্ধতিতে ট্রাভার্স করলাম। তাতেই নোডগুলো ছোটো থেকে বড়ো ক্রমে পেয়ে গেলাম।

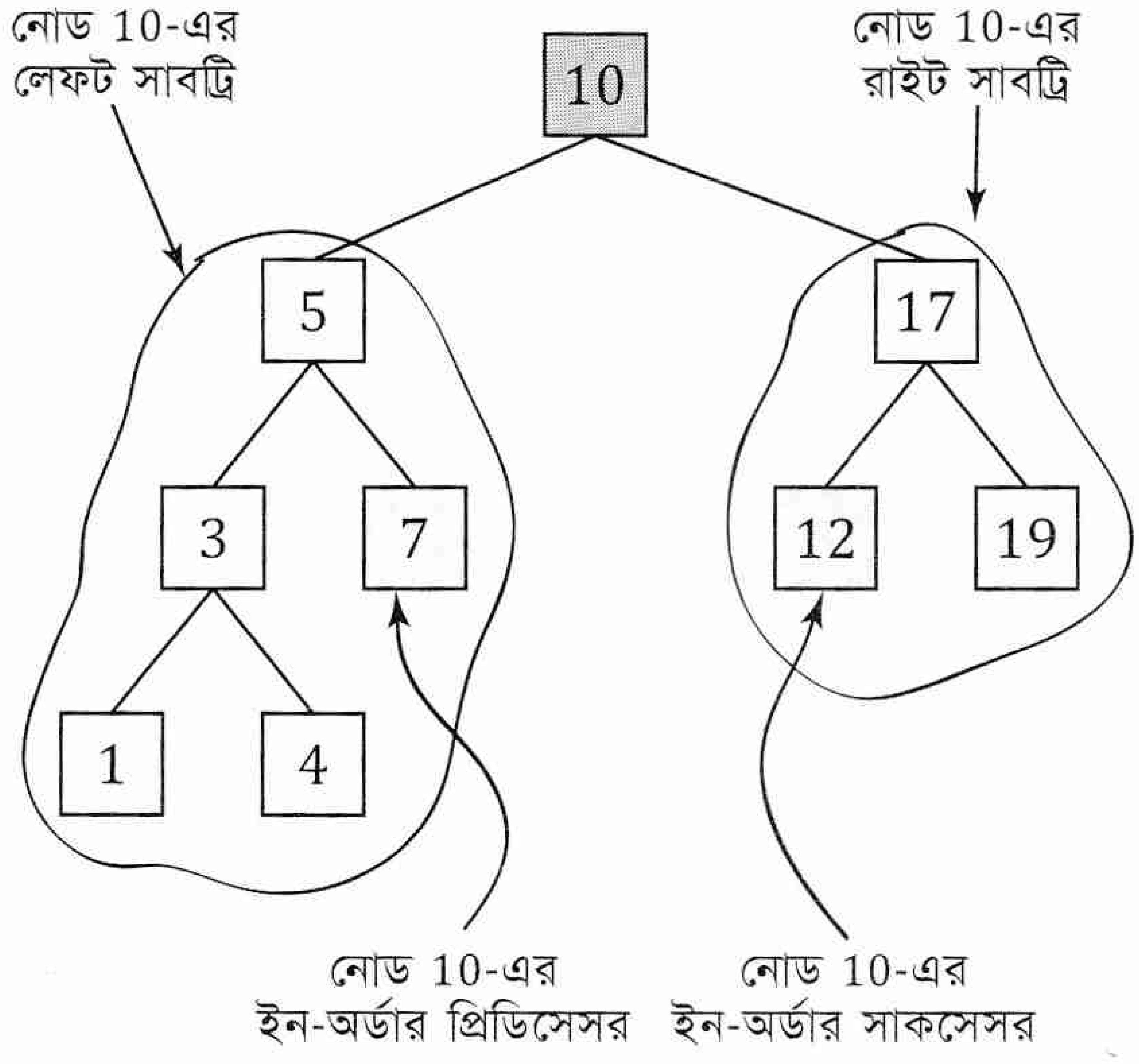
আমরা যদি কোনো বিএসটি-তে ছোটো থেকে বড়ো সাজানো অবস্থায় একটি নোডের আগের নোড ও পরের নোড পেতে চাই, তাহলে কীভাবে পাব? একটি নোডের ডান দিকের সাবট্রি-এর সবচেয়ে ছোটো নোডটি হচ্ছে সেই নোডের পরবর্তী নোড বা উত্তরসূরি (ইংরেজিতে বলে সাকসেসর—successor)। আর একটি নোডের বাঁ দিকের সাবট্রি-এর সবচেয়ে বড়ো নোডটি হচ্ছে সেই নোডের আগের নোড বা পূর্বসূরি (ইংরেজিতে বলে প্রিডিসেসর—predecessor)।

ছবি 8.6-এর ট্রি-টি লক্ষ করো—



ছবি 8.6

এখানে 5-এর উত্তরসূরি বা সাকসেসর হচ্ছে 7, 10-এর উত্তরসূরি হচ্ছে 12। আবার 10-এর পূর্বসূরি হচ্ছে 7, 5-এর পূর্বসূরি হচ্ছে 4 (ছবি 8.7)।

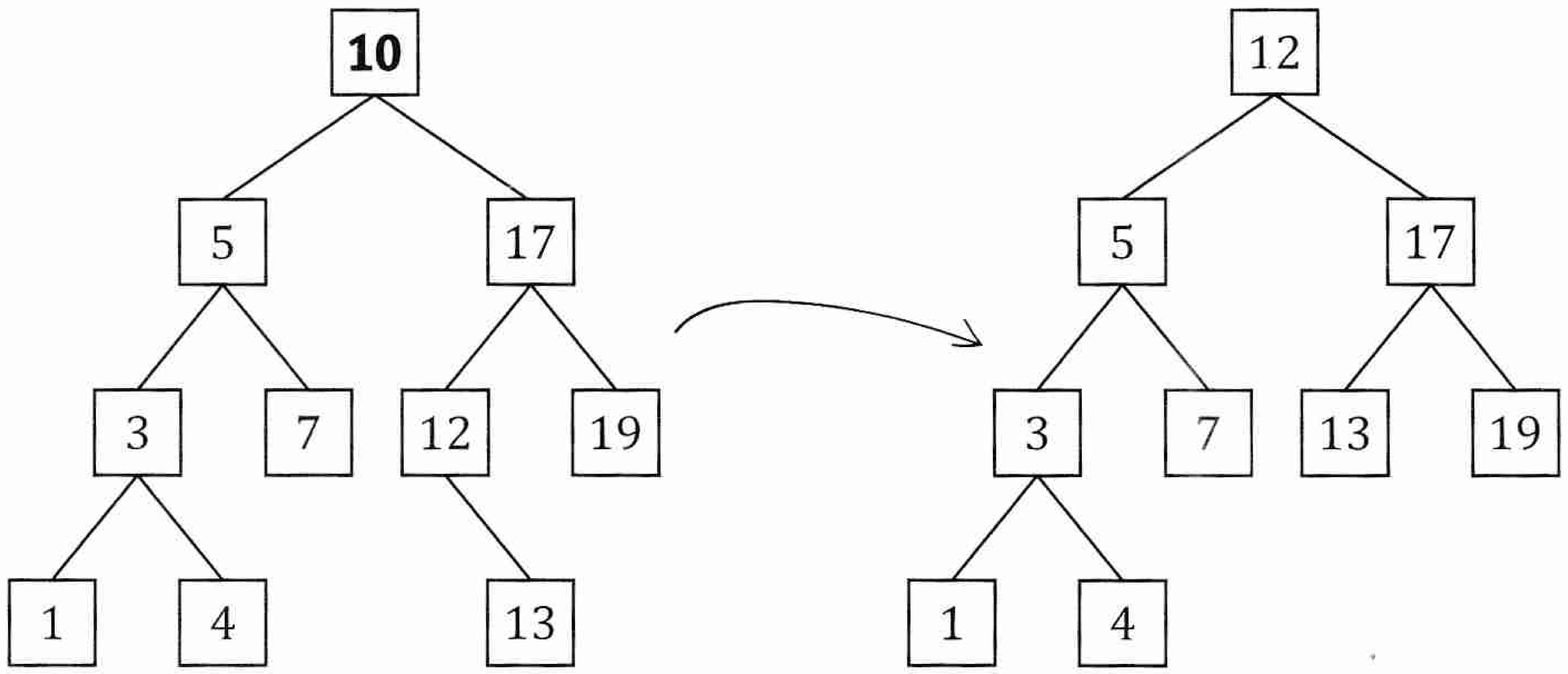


ছবি 8.7

এখন তোমাদের উচিত হবে খাতা-কলম নিয়ে বসা। তারপর নিজে কয়েকটি ট্রি এঁকে পরীক্ষা-নিরীক্ষা করে দেখা।

এখন আমরা যদি বিএসটি থেকে কোনো নোড মুছে ফেলতে চাই, যার দুটি চাইল্ডই আছে, তাহলে সেই নোডের জায়গায় তার পরবর্তী নোড বা উত্তরসূরি নোডকে বসাতে হবে। এখন উত্তরসূরি নোডের যদি আবার চাইল্ড নোড থাকে (চাইল্ড নোড থাকলে কেবল ডান দিকের চাইল্ড নোড থাকবে, বাঁ দিকের চাইল্ড নোড থাকা সম্ভব নয়, কারণ সেটি ইতিমধ্যেই ডান দিকের সাবট্রি-এর সবচেয়ে ছোটো নোড), তাহলে সেই চাইল্ড নোডটি আমরা উত্তরসূরি নোডের জায়গায় বসিয়ে দেব।

এখন আমরা বিএসটি-তে একটি নোডের জায়গায় আরেকটি নোড বসিয়ে দেওয়ার কাজটি করার জন্য একটি ফাংশন লিখে ফেলব। ফাংশনের নাম bst_transplant কেন? কারণ transplant (উচ্চারণ ট্রান্সপ্ল্যান্ট) শব্দের বাংলা অর্থ প্রতিস্থাপন। আর আমরা সেই কাজটিই করতে যাচ্ছি। খেয়াল রাখবে, যখন একটি নোড দিয়ে প্রতিস্থাপন করছি, তখন কিন্তু কেবল সেই নোড নয়, বরং সেই নোড থেকে যেই সাবট্রি শুরু হয়, সেই সাবট্রি আগের নোডের জায়গায় চলে আসবে। অর্থাৎ নতুন নোডটি তার চাইল্ড নোডগুলোসহ চলে আসবে আর কি।



ছবি 8.8: নোড 10-কে বাদ দিতে হলে তার উত্তরসূরি নোড 12-কে তার জায়গায় নিয়ে আসতে হবে। আর নোড 12-এর রাইট সাবট্রি-টি নোড 12-এর জায়গায় বসাতে হবে।

```
Node *bst_transplant(Node *root,
                     Node *current_node,
                     Node *new_node)
{
    if (current_node == root)
    {
        root = new_node;
    }
    else if (current_node == current_node->parent->left)
    {
        add_left_child(current_node->parent, new_node);
    }
    else
    {
        add_right_child(current_node->parent, new_node);
    }

    return root;
}
```

ফাংশনে আমরা বিএসটি-এর current_node নামক নোডের জায়গায় new_node নামক নোড বসিয়ে দেব। ফাংশনের ভেতরে প্রথমে পরীক্ষা করছি যে, current_node আসলে রুট নোড কি না । যদি রুট নোড হয়, তাহলে আমরা সহজেই বলে দিতে পারি যে, এখন রুট নোড হবে new_node। আর যদি তা না হয়, তাহলে আমরা দেখব current_node তার প্যারেন্টের লেফট চাইল্ড নাকি রাইট চাইল্ড। যদি লেফট চাইল্ড হয়, তাহলে সেই প্যারেন্ট নোডের লেফট

চাইল্ড হিসেবে new_node-কে যুক্ত করব। আর যদি রাইট চাইল্ড হয়, তাহলে প্যারেন্ট নোডের রাইট চাইল্ড হিসেবে new_node-কে যুক্ত করে দেব।

ইতিমধ্যেই কিন্তু আমরা বিএসটি থেকে একটি নোড বাদ দেওয়ার জন্য যা যা জানা প্রয়োজন, সেগুলো জেনে ফেলেছি। এখন bst_delete() ফাংশন তৈরি করার পালা।

```
/*
   ফাংশনটির প্রথম প্যারামিটার হচ্ছে ট্রি-এর রুট। দ্বিতীয় প্যারামিটার হচ্ছে একটি নোড,
   যা আমরা ট্রি থেকে মুছে ফেলব।  নোডটি মুছে ফেলার পরে পরিবর্তিত ট্রি-এর রুট নোড
   ফাংশন থেকে রিটার্ন করা হবে।
*/

Node *bst_delete(Node *root, Node *node)
{
    Node *smallest_node;

    if (node->left == NULL) {
        // নোডের বাঁ দিকে কোনো চাইল্ড নেই।
        // তাই ডান দিকের  চাইল্ড দিয়ে নোডকে প্রতিস্থাপন করব।
        root = bst_transplant(root, node, node->right);
    } else if (node->right == NULL) {
        // নোডের ডান দিকে কোনো চাইল্ড নেই।
        // তাই বাঁ দিকের চাইল্ড দিয়ে নোডকে প্রতিস্থাপন করব।
        root = bst_transplant(root, node, node->left);
    } else {
        // node-এর দুই দিকেই (বাঁ ও ডান) চাইল্ড নোড আছে।
        // node-এর ডান দিকের যেই সাবট্রি, সেখান থেকে
        // সবচেয়ে ছোটো নোডটি বের করব।
        smallest_node = bst_minimum(node->right);

        // smallest_node, node-এর সরাসরি চাইল্ড কি না সেটি পরীক্ষা করি
        if (smallest_node->parent != node) {
            // smallest_node, node-এর সরাসরি চাইল্ড নয়,
            // তাই আমাদেরকে দুটি কাজ করতে হবে।
            // smallest_node-কে তার ডান দিকের চাইল্ড দিয়ে প্রতিস্থাপন
            // করি।
            root = bst_transplant(root, smallest_node,
                                  smallest_node->right);
```

```
            // smallest_node-এর রাইট চাইল্ড হিসেবে node-এর রাইট
            // চাইল্ডকে যুক্ত করি।
            add_right_child(smallest_node, node->right);
        }

        // node-কে smallest_node দিয়ে প্রতিস্থাপন করি।
        root = bst_transplant(root, node, smallest_node);

        // আমরা node মুছে দেব, তাই smallest_node-এর লেফট চাইল্ড
        // হিসেবে node-এর লেফট চাইল্ডকে যুক্ত করে দিই।
        add_left_child(smallest_node, node->left);
    }

    // এবারে node-এর জন্য বরাদ্দ মেমোরি ছেড়ে নিই।
    free(node);

    // root রিটার্ন করি।
    return root;
}
```

এখানে প্রথম দুটি কন্ডিশনে আমরা পরীক্ষা করেছি যে, node-এর কেবল একটিমাত্র চাইল্ড আছে কি না। থাকলে সেই অনুযায়ী ব্যবস্থা নিয়েছি (কী ব্যবস্থা সেটি বুঝতে না পারলে আরো কয়েক পৃষ্ঠা আগে থেকে আবার পড়তে হবে)। যদি কোনো চাইল্ডই না থাকে, তাহলেও সমস্যা নেই। তখন প্রথম শর্তটি সত্য হবে এবং node-এর বদলে node->right (যেটি আসলে NULL, কারণ node-এর কোনো চাইল্ড না থাকলে node->left ও node->right দুটোই হবে NULL) বসে যাবে, অর্থাৎ NULL বসে যাবে। তাহলে ট্রি থেকে node মুছে গেল।

আর যদি node-এর দুটো চাইল্ড থাকে, তাহলে আমরা তার ডান দিকের সাবট্রি-এর সবচেয়ে ছোটো নোডটি বের করে min_node-এ রেখেছি। সেই ছোটো নোডটি যদি node-এর সরাসরি চাইল্ড হয়, তাহলে তো node-এর জায়গায় min_node বসিয়ে দিলেই হবে। আর যদি তা না হয়, তাহলে আমাদেরকে প্রথমে কিছু কাজ করতে হবে (যা আমরা করেছি if কন্ডিশনের ভেতরে)। প্রথমে আমরা min_node-এর ডান দিকের চাইল্ডকে min_node-এর জায়গায় বসাব (ডান দিকে কোনো চাইল্ড না থাকলে NULL বসবে)। তারপরে node-এর রাইট নোডকে min_node-এর রাইট নোড হিসেবে যুক্ত করব। তারপরে আমরা node-কে min_node দিয়ে বদলে দেব। সবশেষে root নোডটি রিটার্ন করব। কারণ কোনো কারণে যদি রুট নোডকে বাদ

দিতে হয়, তাহলে তো এখন রুট হিসেবে অন্য একটি নোড আছে। তাই রুট নোড পরিবর্তিত হওয়ার সম্ভাবনা সৃষ্টি হয় বলেই আমরা root নোড রিটার্ন করব।।

> বাইনারি সার্চ ট্রি-এর এই নোড মুছে ফেলার কাজটি অপেক্ষাকৃত জটিল, কারণ এখানে বিভিন্ন রকম পরিস্থিতি চিন্তা করতে হয়। তাই ঠিকভাবে বুঝতে হলে বাইনারি সার্চ ট্রি পুরো অংশটুকু একাধিক বার পড়তে হতে পারে।

এখন আমরা bst_delete() ফাংশনটি পরীক্ষা করব।

```
int main()
{
    Node *root = create_bst();
    Node *node;

    printf("BST:\n");
    in_order(root);
    printf("\n");

    node = bst_search(root, 1);

    if (node != NULL) {
        printf("Will delete %d\n", node->data);
        root = bst_delete(root, node);

        printf("BST:\n");
        in_order(root);
        printf("\n");

    } else {
        printf("Node not found!\n");
    }

    node = bst_search(root, 5);

    if (node != NULL) {
        printf("Will delete %d\n", node->data);
        root = bst_delete(root, node);

        printf("BST:\n");
        in_order(root);
```

```
        printf("\n");

    } else {
        printf("Node not found!\n");
    }

    node = bst_search(root, 10);

    if (node != NULL) {
        printf("Will delete %d\n", node->data);
        root = bst_delete(root, node);

        printf("BST:\n");
        in_order(root);
        printf("\n");

    } else {
        printf("Node not found!\n");
    }

    return 0;
}
```

প্রোগ্রামটি রান করলে আউটপুট আসবে এরকম—

```
BST:
1  3  4  5  7  10  12  17  19
Will delete 1
BST:
3  4  5  7  10  12  17  19
Will delete 5
BST:
3  4  7  10  12  17  19
Will delete 10
BST:
3  4  7  12  17  19
```

আউটপুট দেখে আমরা বুঝতে পারছি যে, ট্রি থেকে নোড মুছে যাচ্ছে। এখন তাহলে ছবি এঁকে দেখা যাক।

```
        _10_
       /    \
      5      17
     / \    /  \
    3   7  12   19
   / \
  1   4
```

ওপরের ট্রি থেকে 1 বাদ দিলে ট্রি-এর অবস্থা দাঁড়াবে নিচের মতো :

```
        _10_
       /    \
      5      17
     / \    /  \
    3   7  12   19
     \
      4
```

ওপরের ট্রি থেকে 5 বাদ দিলে ট্রি-এর অবস্থা দাঁড়াবে নিচের মতো :

```
        _10_
       /    \
      7      17
     /      /  \
    3      12   19
     \
      4
```

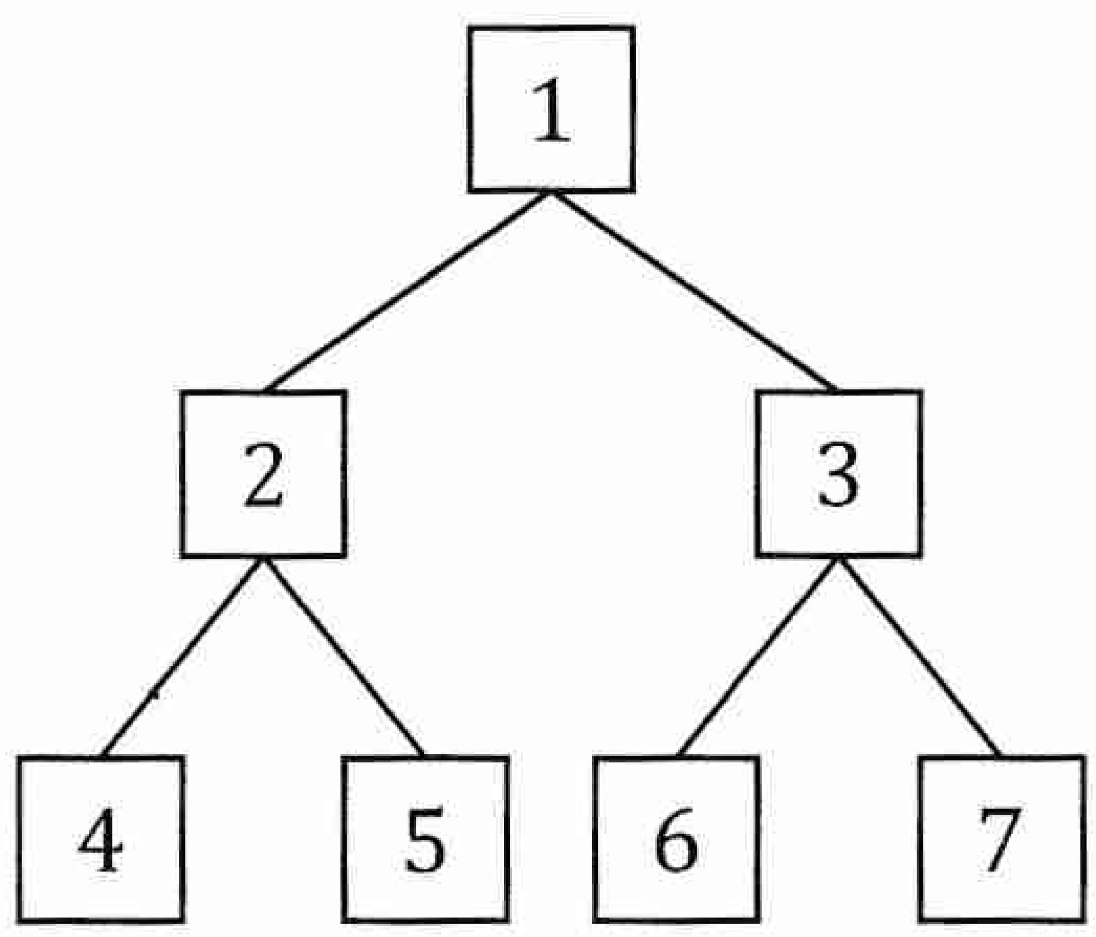
ওপরের ট্রি থেকে 10 বাদ দিলে ট্রি-এর অবস্থা দাঁড়াবে নিচের মতো :

```
        _12_
       /    \
      7      17
     /         \
    3           19
     \
      4
```

অধ্যায় ৯ : হিপ, হিপ সর্ট ও প্রায়োরিটি কিউ

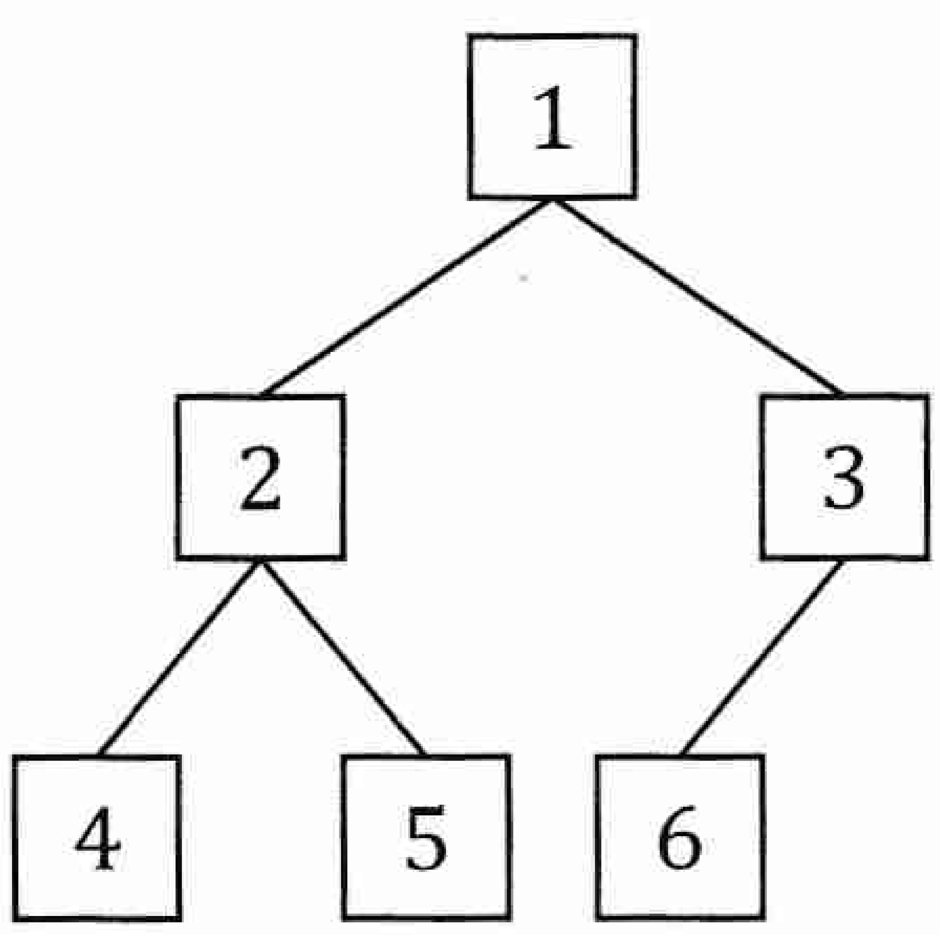
হিপ

আমরা ইতিমধ্যেই বাইনারি ট্রি সম্পর্কে জেনেছি। এখন, কোনো বাইনারি ট্রি-এর লিফ নোড বাদে বাকি সবগুলো নোডের যদি দুটি করে চাইল্ড থাকে, তাকে বলে ফুল বাইনারি ট্রি (full binary tree)। যেমন, ছবি 9.1-এর ট্রি-টি হচ্ছে একটি ফুল বাইনারি ট্রি।

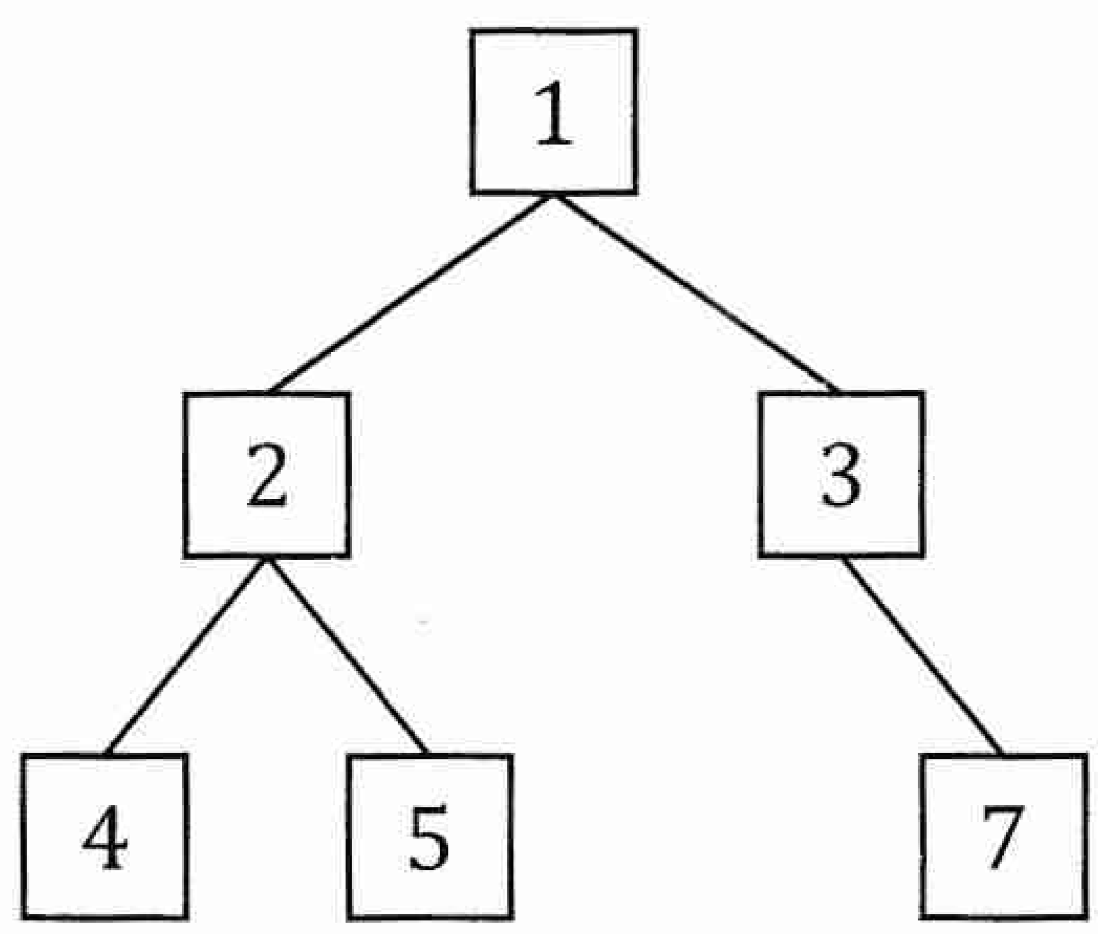


ছবি 9.1

আর যদি কোনো বাইনারি ট্রি এমন হয় যে, তার শেষ লেভেলের আগের লেভেল পর্যন্ত ট্রি-টি ফুল বাইনারি ট্রি এবং শেষ লেভেলের নোডগুলো বাঁ দিকের ঘরগুলো পূরণ করে ক্রমশ ডান দিকে আসে, সেই ট্রি-কে বলা হয় কমপ্লিট বাইনারি ট্রি (complete binary tree)। যেমন, ছবি 9.2-এর ট্রি-টি হচ্ছে একটি কমপ্লিট বাইনারি ট্রি।



ছবি 9.2



ছবি 9.3

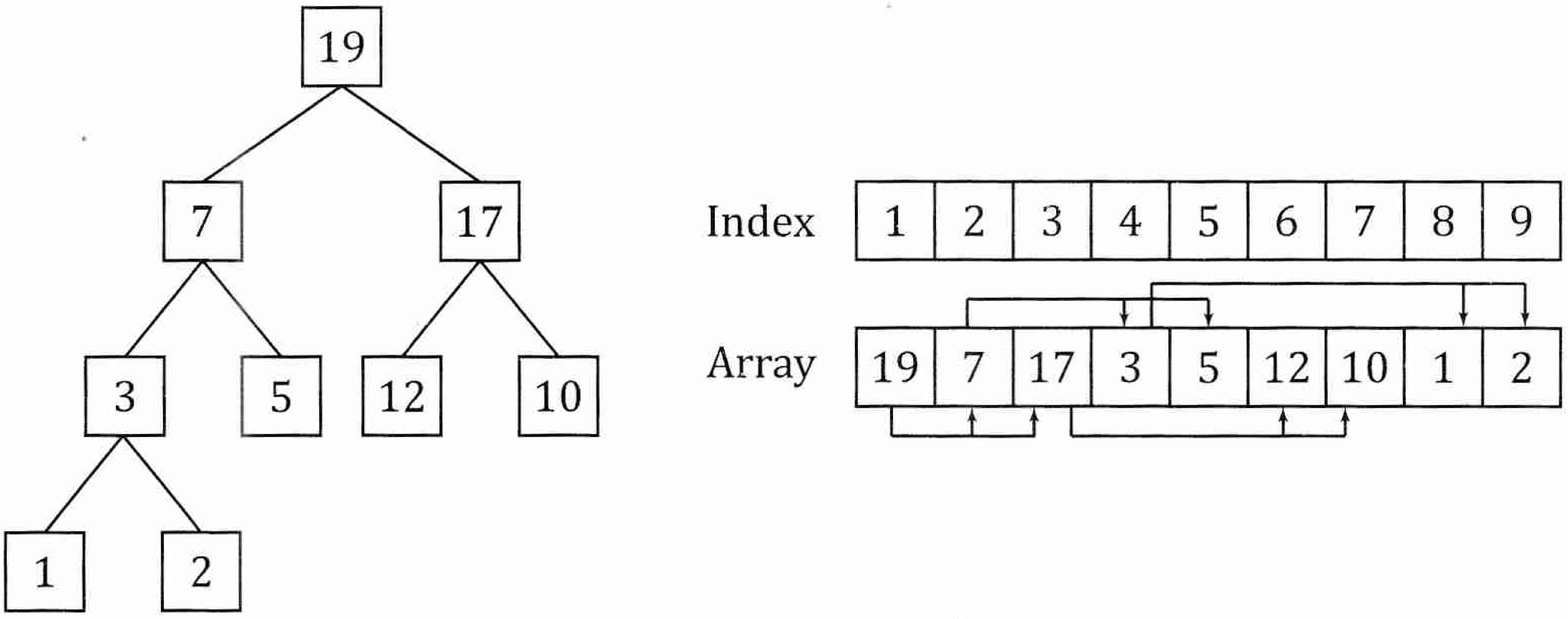
কিন্তু ছবি 9.3-এ যেই ট্রি দেখানো হয়েছে, সেটি কমপ্লিট বাইনারি ট্রি নয়। কারণ 7-এর আগের ঘরটি (3-এর লেফট চাইল্ড) ফাঁকা আছে।

হিপ (heap) হচ্ছে কমপ্লিট বাইনারি ট্রি যেখানে প্রতিটি নোডের মান তার চাইল্ড নোড দুটির মানের চেয়ে বড়ো (ম্যাক্স-হিপ) বা ছোটো (মিন-হিপ)। উল্লেখ্য যে, হিপ শব্দের আভিধানিক অর্থ হলো স্তূপ, কিন্তু এখানে হিপ বলতে আমরা হিপ নামক ডেটা স্ট্রাকচারকেই বুঝব। একটি হিপের প্রতিটি নোডের মান যদি তার চাইল্ড নোডদ্বয়ের মানের চেয়ে বড়ো হয়, সেই হিপ-কে বলা হয় ম্যাক্স হিপ (max heap)। আর প্রতিটি নোডের মান যদি তার চাইল্ড নোডদ্বয়ের মানের চেয়ে ছোটো হয়, সেই হিপ-কে বলা হয় মিন হিপ (min heap)। আমরা এই বইতে ম্যাক্স হিপ নিয়েই আলোচনা করব এবং যখন হিপ লিখব, তখন ম্যাক্স হিপকেই বোঝাব। ম্যাক্স হিপ ঠিকভাবে বুঝতে পারলে মিন হিপও সহজে বুঝতে পারা যাবে। ম্যাক্স হিপের মূলনীতি হচ্ছে, ট্রি-এর যেকোনো নোডের মান, তার চাইল্ড নোডগুলোর মানের চেয়ে বড়ো কিংবা সমান। আমরা হিপের মূলনীতি বলতে এখন থেকে (কেবল এই বইতে) ম্যাক্স হিপের মূলনীতিকেই বোঝাব।

অনুশীলনী :

মিন হিপের মূলনীতি কী?

হিপ যেহেতু কমপ্লিট বাইনারি ট্রি, তাই আমরা একে মেমোরিতে রাখার জন্য অ্যারে ব্যবহার করতে পারি। অ্যারের n-তম ঘরে যদি একটি নোড থাকে, তার লেফট চাইল্ড থাকবে $(2 \times n)$-তম ঘরে আর রাইট চাইল্ড থাকবে $(2 \times n + 1)$-তম ঘরে। আর যেকোনো নোডের প্যারেন্ট নোডের ইনডেক্স পেতে চাইলে তার ইনডেক্সকে 2 দিয়ে ভাগ করলেই হবে। হিসাবনিকাশের সুবিধার জন্য আমরা হিপের প্রথম নোডকে অ্যারের 0-তম ইনডেক্সে না রেখে 1-তম ইনডেক্স রাখব এবং বাকি ঘরগুলোতে বাকি নোডগুলো রাখব। এখন আমরা দেখি নিচের হিপটি একটি অ্যারেতে রাখলে, কোন ঘরে কোন নোডটি রাখতে হবে?



ছবি 9.4

হিপটি রাখার জন্য আমরা heap নামে একটি অ্যারে তৈরি করব। বর্তমানে হিপটিতে নয়টি নোড রয়েছে। আমরা যদি জানি যে, হিপটিতে সর্বোচ্চ কয়টি নোড থাকবে, তাহলে আমরা সেই অনুযায়ী অ্যারের সাইজ ঠিক করে দিতে পারি। এই উদাহরণের জন্য আমরা 10 সাইজের একটি অ্যারে তৈরি করব (কারণ আমরা প্রথম উপাদান, অর্থাৎ রুট নোডকে `heap[0]`-তে না রেখে `heap[1]`-এ রাখব)।

```
int heap[10];
```

হিপের রুট-কে আমরা রাখব `heap[1]`-এ। রুট নোডের দুটি চাইল্ড হচ্ছে 7 ও 17। এদেরকে যথাক্রমে $1 \times 2 = 2$ এবং $1 \times 2 + 1 = 3$ নম্বর ঘরে রাখব।

```
heap[1] = 19;
heap[2] = 7;
heap[3] = 17;
```

এখন, নোড 7 আছে 2 নম্বর ঘরে। তাই এর লেফট চাইল্ড থাকবে— $2 \times 2 = 4$ নম্বর ঘরে এবং রাইট চাইল্ড থাকবে—$2 \times 2 + 1 = 5$ নম্বর ঘরে। তেমনি, নোড 17 আছে 3 নম্বর ঘরে, তাই এর লেফট চাইল্ড থাকবে— (3×2) বা 6 নম্বর ঘরে ও রাইট চাইল্ড থাকবে— $(3 \times 2 + 1)$ বা 7 নম্বর ঘরে।

```
heap[2*2] = 3;
heap[2*2 + 1] = 5;
heap[3*2] = 12;
heap[3*2 + 1] = 10;
```

তেমনি, নোড 3 আছে 4 নম্বর ঘরে। তাহলে তার চাইল্ডগুলোকেও আমরা যথাস্থানে রাখতে পারি।

```
heap[4*2] = 1;
heap[4*2 + 1] = 2;
```

এখন, যদি জানতে চাওয়া হয়, নোড 17-এর রাইট চাইল্ড কে? সেজন্য আমাদেরকে নোড 17-এর ইনডেক্স জানতে হবে। নোড 17-এর ইনডেক্স হচ্ছে 3 (অর্থাৎ, 17 আছে 3 নম্বর ঘরে— ছবি 9.4)। তাই এর রাইট চাইল্ড পাওয়া যাবে— $(3 \times 2 + 1)$ বা 7 নম্বর ঘরে। সেই ঘরে আছে 10। তাই 17-এর রাইট চাইল্ড হচ্ছে 10। আবার, নোড 1-এর প্যারেন্ট কে? নোড 1-এর ইনডেক্স হচ্ছে 8। তাই এর প্যারেন্ট থাকার কথা— $\frac{8}{2} = 4$ নম্বর ঘরে। 4 নম্বর ঘরে আছে 3। তাই 1-এর

প্যারেন্ট হচ্ছে 3। তেমনি নোড 5-এর ইনডেক্স হচ্ছে 5। তাই এর প্যারেন্ট থাকবে— $\frac{5}{2} = 2$ নম্বর ঘরে (এখানে আমরা ভাগফলের কেবল ইন্টিজার অংশটুকু নেব)। তাই 5-এর প্যারেন্ট হচ্ছে 7।

তাহলে আমরা নিচের ফাংশনগুলো লিখে ফেলতে পারি—

```
int left(int i)
{
    return 2 * i;
}

int right(int i)
{
    return 2 * i + 1;
}

int parent(int i)
{
    return i / 2;
}
```

ফাংশনগুলো কী কাজ করছে, তা এতক্ষণ যেই আলোচনা করা হয়েছে, সেটুকু ঠিকভাবে পড়লে বুঝতে অসুবিধা হওয়ার কথা নয়। তাহলে একটি অ্যারে দেওয়া থাকলে সেটি ম্যাক্স হিপ কি না, সেটি বের করার জন্য একটি ফাংশন লিখে ফেলি।

```
int is_max_heap(int H[], int heap_size)
{
    int i, p;

    for (i = heap_size; i > 1; i--) {
        p = parent(i);
        // প্রোগ্রামটা কীভাবে কাজ করে, সেটি বুঝতে সুবিধা হবে,
        // যদি তুমি নিচের লাইনের কমেন্ট চিহ্ন (//) মুছে দাও।
        //printf("i = %d, p = %d\tH[i] = %d, H[p] = %d\n",
                i, p, H[i], H[p]);

        if (H[p] < H[i]) {
            // প্যারেন্টের মান যদি চাইল্ডের মানের চেয়ে ছোটো হয়, তাহলে সেটি
            // ম্যাক্স হিপ নয়। তাই 0 রিটার্ন করে ফাংশন থেকে বের হয়ে যাচ্ছি।
            return 0;
```

```
        }
    }

    // প্রোগ্রাম এই পর্যন্ত আসার মানে হচ্ছে, এমন কোনো নোড পাওয়া যায়নি,
    // যার প্যারেন্টের মান ওই নোডের চেয়ে ছোটো, অর্থাৎ সবজায়গায় ম্যাক্স হিপ
    // প্রোপার্টি ঠিক আছে। তাই আমরা বলতে পারি, H একটি ম্যাক্স হিপ এবং ফাংশন
    // থেকে 1 রিটার্ন করে সেটি জানিয়ে দিচ্ছি।
    return 1;
}
```

এখন মেইন ফাংশন থেকে ওপরের ফাংশনটি কল করে দেখতে হবে। মেইন ফাংশনের কোড নিজেরা লিখে ফেলো। তোমরা নিচের দুটি হিপ ব্যবহার করে আউটপুট পর্যবেক্ষণ করতে পারো–

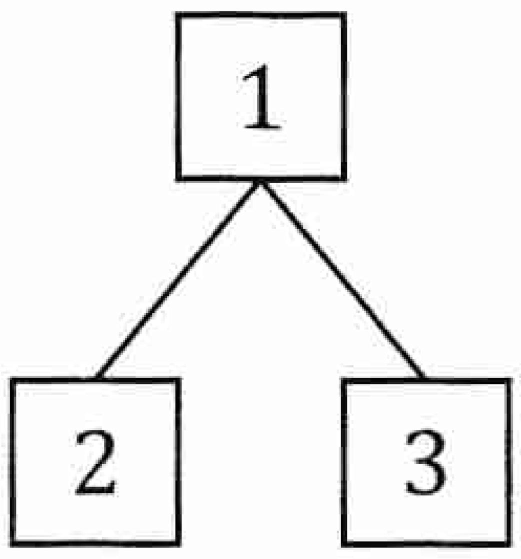
```
int heap_size = 9;
int H[] = {0, 19, 7, 17, 3, 5, 12, 10, 1, 2};
```

এবং

```
int heap_size = 9;
int H[] = {0, 19, 7, 17, 3, 5, 12, 10, 1, 4};
```

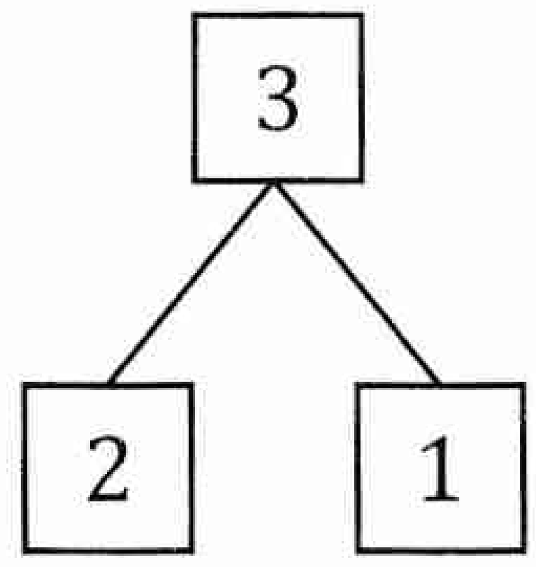
এখন আমরা দেখব, একটি কমপ্লিট বাইনারি ট্রি দেওয়া হলে, আমরা সেটিকে কীভাবে হিপ-এ পরিণত করতে পারি। হিপ-এ পরিণত করতে হলে আমাদেরকে নিশ্চিত করতে হবে যে, প্রতিটি নোডের মান তার চাইল্ড নোড দুটির মানের চেয়ে বড়ো (কিংবা সমান)। অর্থাৎ, প্রতিটি নোড হিপের মূলনীতি অনুসরণ করে।

ধরা যাক, আমাকে ছবি 9.5-এর ট্রি-টি দেওয়া হলো–



ছবি 9.5

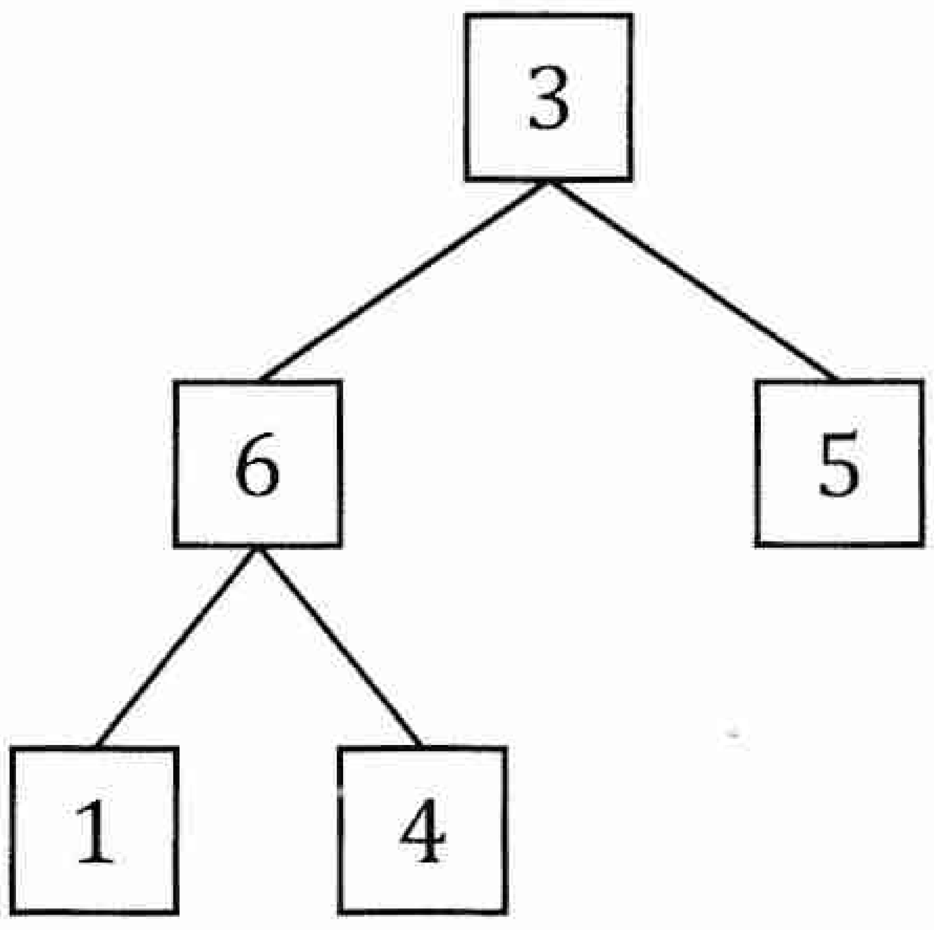
এখন আমাকে দেখতে হবে, 1-এর যে দুটি চাইল্ড রয়েছে, তাদের মধ্যে কে সবচেয়ে বড়ো? উত্তর হচ্ছে, 3। তাহলে 1 আর 3-এর মধ্যে জায়গা অদলবদল করতে হবে (ছবি 9.6)।



ছবি 9.6

আরো বড়ো ট্রি-এর বেলায় কীভাবে কাজটি করব? আমাদেরকে প্রথমে নিশ্চিত করতে হবে, ট্রি-এর সবচেয়ে নিচের লেভেলের সাবট্রিগুলো হিপ-এর মূলনীতি অনুসরণ করে। অর্থাৎ, নোডগুলোর মান তাদের চাইল্ডগুলোর মানের চেয়ে বড়ো। এখন সবচেয়ে নিচের লেভেলে কেবল লিফ নোড আছে, যাদের আর কোনো চাইল্ড নেই। তাহলে তাদেরকে আমরা একেকটি হিপ বলতে পারি। তারপরে তার ওপরের লেভেলে যাব, যেখান প্রতিটি নোডের সর্বোচ্চ দুটি চাইল্ড থাকবে। তো সেই নোডগুলোর বাঁ দিকের সাবট্রি ও ডান দিকের সাবট্রি কিন্তু ইতিমধ্যে হিপ (কারণ তারা হিপের মূলনীতি অনুসরণ করে)। এখন আমরা ওই লেভেলের সবগুলো নোডের ক্ষেত্রে পরীক্ষা করব যে, তারা হিপের মূলনীতি অনুসরণ করে কি না, আর যদি না করে, সে ক্ষেত্রে কী করতে হবে, তা একটু আগেই ছবির মাধ্যমে বোঝানো হয়েছে।

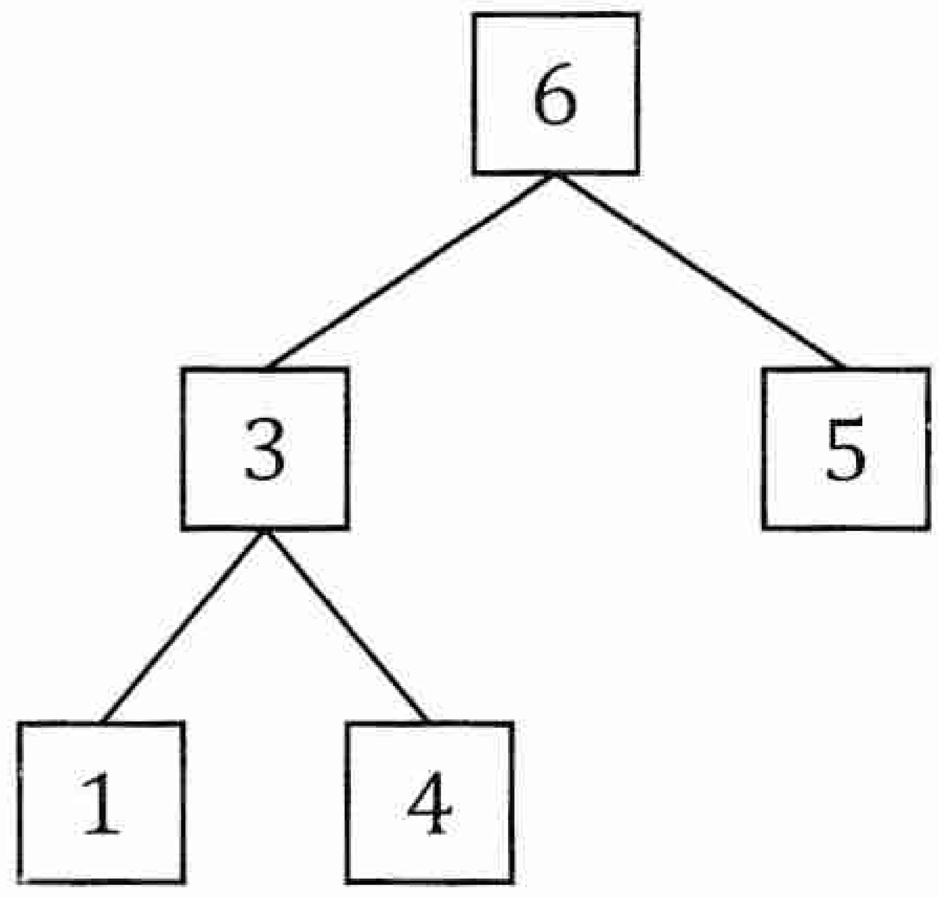
এখন আমরা ওপরের পদ্ধতি ব্যবহার করে একটি উদাহরণ দেখব। ছবি 9.7 লক্ষ করি—



ছবি 9.7

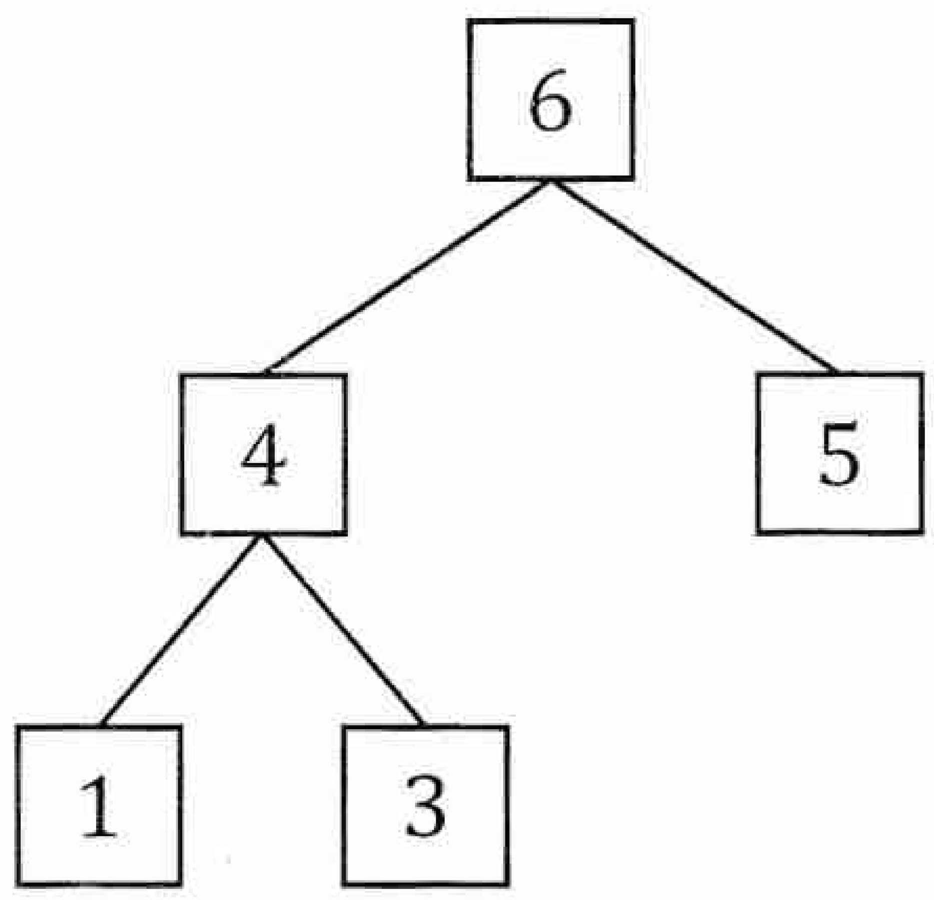
এখানে, নোড 3-এর বাঁ দিকের সাবট্রি একটি হিপ, ডান দিকের সাবট্রিও একটি হিপ। এখন আমাদেরকে এমন কিছু করতে হবে যেন, পুরো ট্রি-এর সবগুলো নোড হিপের মূলনীতি অনুসরণ করে। অর্থাৎ ট্রি-টি যেন একটি হিপ-এ পরিণত হয়ে যায়।

এজন্য প্রথমে দেখব, 3 এবং তার চাইল্ডগুলোর মধ্যে বৃহত্তর কে? 3, 6, 5-এর মধ্যে বৃহত্তর হচ্ছে 6। তারপর তাই 3-এর সঙ্গে 6-এর জায়গা অদলবদল করতে হবে (ছবি 9.8)।



ছবি 9.8

এখন 3 দিয়ে যে সাবট্রি শুরু হলো, সেখানেও একই কাজ করতে হবে (রিকার্শন!)। 3-এর চাইল্ড হচ্ছে 1 ও 4। 3, 1, 4-এর মধ্যে বৃহত্তর হচ্ছে 4। তাই 3 ও 4-এর মধ্যে জায়গা অদলবদল করতে হবে (ছবি 9.9)।



ছবি 9.9

এখন পুরো ট্রি-টি একটি হিপে পরিণত হয়েছে। প্রতিটি নোডই হিপের মূলনীতি অনুসরণ করছে। এই কাজটি করার জন্য এখন একটি ফাংশন লিখে ফেলি। ফাংশনটি ইনপুট হিসেবে একটি হিপ (আসলে অ্যারে) ও একটি ভ্যারিয়েবল নেবে। ভ্যারিয়েবলটি একটি ইনডেক্স নির্দেশ করবে। আমাদের কাজ হবে, ওই ইনডেক্সকে রুট ধরে যেই ট্রি (বা সাবট্রি) আছে, সেটি যেন একটি হিপ হয়, তা নিশ্চিত করা। তবে আমাদের কাজটি সহজ এজন্য যে, ওই ইনডেক্সকে রুট ধরে যেই ট্রি-টি আছে, তার দুটি সাবট্রির প্রতিটি ইতিমধ্যেই হিপ। যেমন আগের উদাহরণে, 3-এর বাঁ দিক ও ডান দিকের সাবট্রি ছিল হিপ। ফাংশনটির নামকরণ কী করব? যেকোনো নামই রাখা যায়, তবে কোরম্যানের বই অনুসারে আমরা ফাংশনটির নাম রাখতে পারি max_heapify। আর এই ফাংশনেও কিন্তু আমাদেরকে রিকার্শন ব্যবহার করতে হবে।

```
/*  ফাংশনে আমরা তিনটি প্যারামিটার ব্যবহার করেছি–
      প্রথমটি হচ্ছে হিপ রাখার জন্য ব্যবহৃত অ্যারে
      দ্বিতীয় প্যারামিটার হচ্ছে হিপের মোট উপাদান সংখ্যা,
      তৃতীয় প্যারামিটার হচ্ছে নোডের ইনডেক্স,  যেখান থেকে ফাংশনটি কাজ শুরু করবে।
```

```
    ফাংশনটি কোনো কিছু রিটার্ন করবে না।
*/

void max_heapify(int heap[], int heap_size, int i)
{
    int l, r, largest, t;

    l = left(i);  // বাঁ দিকের চাইল্ডের ইনডেক্স l
    r = right(i); // ডান দিকের চাইল্ডের ইনডেক্স r

    if (l <= heap_size && heap[l] > heap[i]) {
        largest = l; // যেহেতু heap[l] বড়ো, তাই largest হবে l
    } else {
        largest = i; // নইলে largest-এর মান হবে i
    }

    // i ও l-এর মধ্যে সবচেয়ে বড়োটি আছে largest-এ। তাই এখন
    // আমরা largest-এর সঙ্গে r-এর তুলনা করব এবং heap[r] যদি
    // heap[largest]-এর চেয়ে বড়ো হয়, তাহলে, largest-এ r
    // অ্যাসাইন করব, না হলে কিছুই করব না (কারণ largest-ই সবচেয়ে বড়ো)।
    if (r <= heap_size && heap[r] > heap[largest]) {
        largest = r;
    }

    // এভাবে আমরা নিশ্চিত করছি যে, heap[i], heap[l] ও heap[r]-এর
    // মধ্যে যার মান সবচেয়ে বড়ো, তার ইনডেক্স largest-এ আছে।
    if (largest != i) {
        // largest যদি i-এর সমান না হয়, তার মানে হচ্ছে, heap[i]-এর
        // কোনো চাইল্ড heap[i]-এর চেয়ে বড়ো, সেই চাইল্ড আছে
        // heap[largest]-এ। তাই আমরা heap[largest]-কে heap[i]-
        // এর জায়গায় নিয়ে আসব, আর heap[i]-কে রাখব
        // heap[largest]-এর (মানে নোড দুটি অদলবদল করব আর কি)।
        t = heap[i];
        heap[i] = heap[largest];
        heap[largest] = t;

        // আমরা i ইনডেক্সে সবচেয়ে বড়ো নোডকে রেখে দিয়েছি, আর largest
        // ইনডেক্সে heap[i]-কে রেখেছি। তাই আমরা আবার max_heapify()
```

```
        // ফাংশন কল করব, তবে এবারে ইনডেক্স হিসেবে largest ব্যবহার করব
        // (যেহেতু সেখানেই heap[i] নোডটি আছে)।
        max_heapify(heap, heap_size, largest);
    }
}
```

এখানে আমরা যে কাজটি করেছি, তা হলো, নোড i-এর বাঁ দিকের ও ডান দিকের নোডের ইনডেক্স বের করেছি, যথাক্রমে– l ও r। এখন, heap[i], heap[l] ও heap[r]-এর মধ্যে সবচেয়ে বড়ো নোডের ইনডেক্সকে largest নামে একটি ভ্যারিয়েবলে রেখেছি। এখন, largest-এর মান যদি i-এর মানের সমান হয়, তার অর্থ হচ্ছে, i-তম নোডটির মান (অর্থাৎ, i-তম ইনডেক্সের ডেটা) তার দুটি চাইল্ডের মানের চেয়ে বড়ো। তাই আমাদের এ ক্ষেত্রে কিছু করতে হবে না। এটিই হচ্ছে রিকার্শনের বেইজ কন্ডিশন। আর যদি তা না হয়, তাহলে আমরা i-তম ইনডেক্সের নোডের সঙ্গে largest ইনডেক্সের নোড অদলবদল করেছি, তারপর largest ইনডেক্স কে রুট নোড ধরে যেই সাবট্রি-টি পাওয়া যাবে, তার জন্য আবার max_heapify(heap, heap_size, largest) ফাংশন কল করেছি।

👁 কারো কারো কাছে কোড দেখে রিকার্শনের বেজ কন্ডিশন বুঝতে কষ্ট হতে পারে। আসলে শেষ কয়েকটি লাইনকে এভাবেও লেখা যায়–

```
if (largest == i) {
    return;
}

t = heap[i];
heap[i] = heap[largest];
heap[largest] = t;

max_heapify(heap, heap_size, largest);
```

এখন max_heapify() ফাংশনটি পরীক্ষা করে দেখা যাক।

```
void print_heap(int heap[], int heap_size)
{
    int i;

    for (i = 1; i <= heap_size; i++) {
        printf("%d ", heap[i]);
    }
```

```
    printf("\n\n");
}

int main()
{
    int heap_size = 9;
    int heap[] = {0, 19, 7, 12, 3, 5, 17, 10, 1, 2};

    // maxi_heapify() ফাংশন কল করার আগে হিপ প্রিন্ট করে দেখি।
    print_heap(heap, heap_size);

    max_heapify(heap, heap_size, 3);

    // maxi_heapify() ফাংশন কল করার পরে হিপ প্রিন্ট করে দেখি।
    print_heap(heap, heap_size);

    return 0;
}
```

প্রোগ্রাম রান করে দেখতে হবে আউটপুট কী আসে। তারপর সেটি নিয়ে চিন্তা করতে হবে যে, কীভাবে কী হলো। প্রয়োজনে খাতা-কলম ব্যবহার করো।

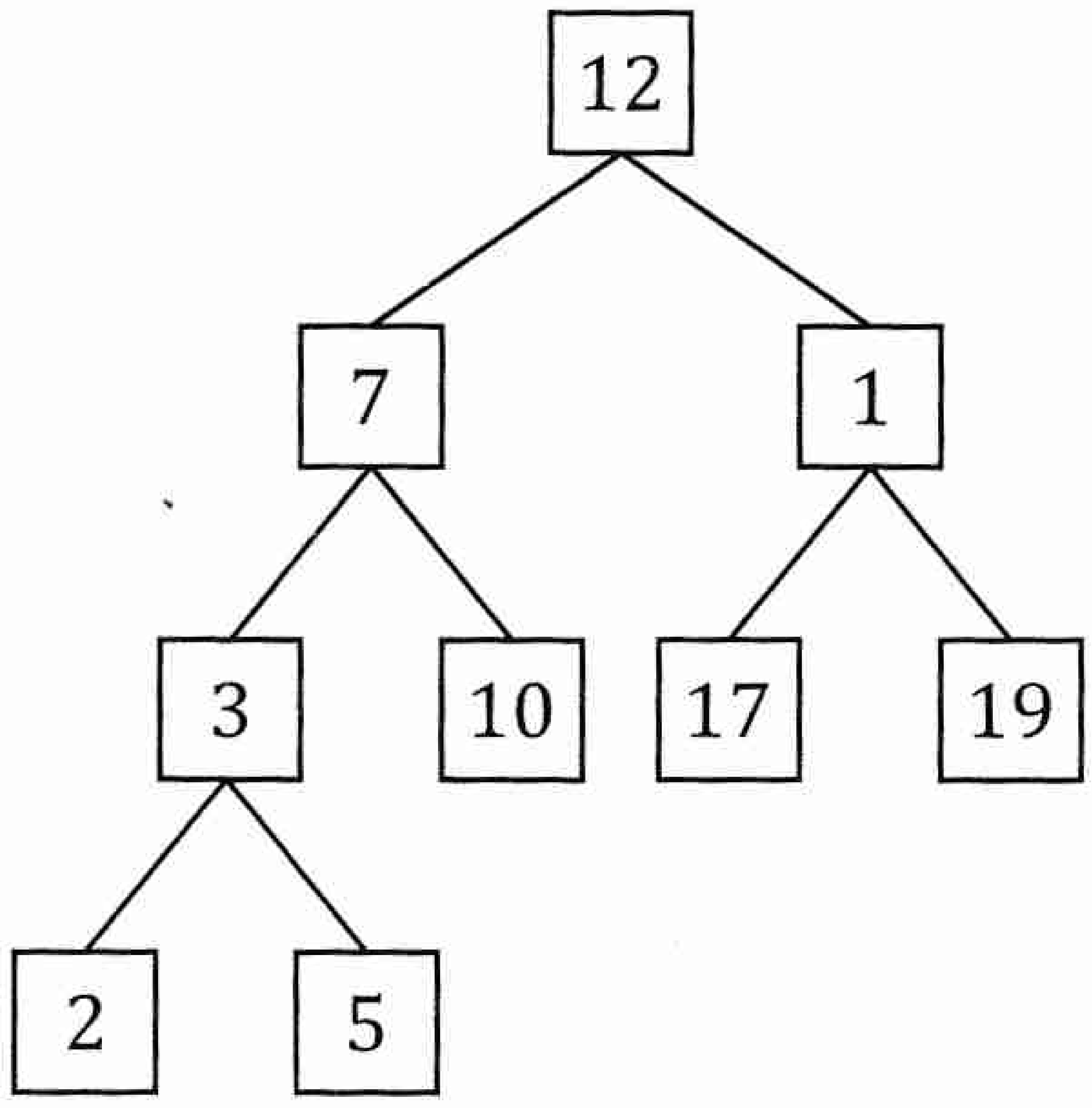
আমাদের কিন্তু হিপ তৈরির কাজ শেষ হয়ে যায়নি। বরং, আমরা কেবল max_heapify() ফাংশনটি তৈরি করলাম, যেখানে একটি নোডের ইনডেক্স পাঠালে, নোডের সাবট্রিগুলো যদি হিপ হয়, তাহলে ফাংশনটি নিশ্চিত করবে যে, ওই নোড এবং তার সাবট্রিগুলো মিলে যে ট্রি তৈরি হয়, সেটিও একটি হিপ। আমরা যেহেতু জানি, যেসব নোডের আর কোনো চাইল্ড নেই, তাদেরকে একটি করে সাবট্রি বিবেচনা করলে তারা প্রত্যেকেই একটি করে হিপ, আমরা সেগুলোর জন্য আলাদা কিছু করব না। যেসব নোডের কমপক্ষে একটি চাইল্ড আছে, সেই নোড থেকে শুরু করে একই লেভেলে ট্রি-এর বাঁ দিক বরাবর এবং তার ওপরের লেভেলগুলোতে যতগুলো নোড আছে (ডান-থেকে-বামে এই ক্রমে), সেগুলোর জন্য max_heapify() ফাংশন কল করব। এভাবে উলটো পথে এগোনোর কারণ হচ্ছে, যখন একটি নোডের জন্য ওই ফাংশনটি আমরা কল করি, আমরা নিশ্চিত হতে চাই যে, ওই নোডের লেফট চাইল্ড ও রাইট চাইল্ডে যে সাবট্রি রয়েছে, তারা ইতিমধ্যে হিপ। আমরা এখন হিপ তৈরি করার ফাংশন লিখে ফেলি।

```
void build_max_heap(int heap[], int heap_size)
{
    int i;
```

```
    for (i = heap_size / 2; i >= 1; i--) {
        max_heapify(heap, heap_size, i);
    }
}
```

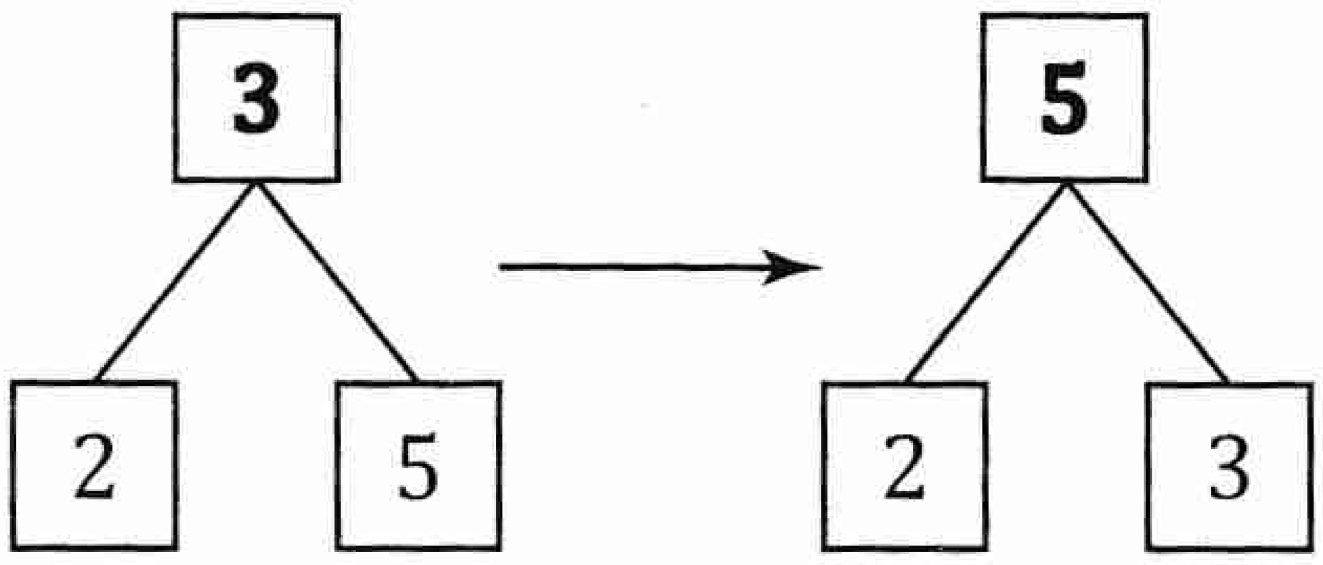
ওপরের কোডে আমরা লুপের ভেতরে i-এর মান শুরু করেছি heap_size-এর অর্ধেক থেকে। কারণ হিপের মধ্যে nসংখ্যক নোড থাকলে $\frac{n}{2}$সংখ্যক নোড হচ্ছে লিফ, তাদের আর কোনো চাইল্ড নোড নেই। তাই সেগুলোর জন্য max_heapify() ফাংশন কল করাটা অপ্রয়োজনীয়।

ধরা যাক, একটি কমপ্লিট বাইনারি ট্রি দেওয়া আছে, এটি থেকে একটি হিপ (ম্যাক্স হিপ) তৈরি করতে হবে (ছবি 9.10)।



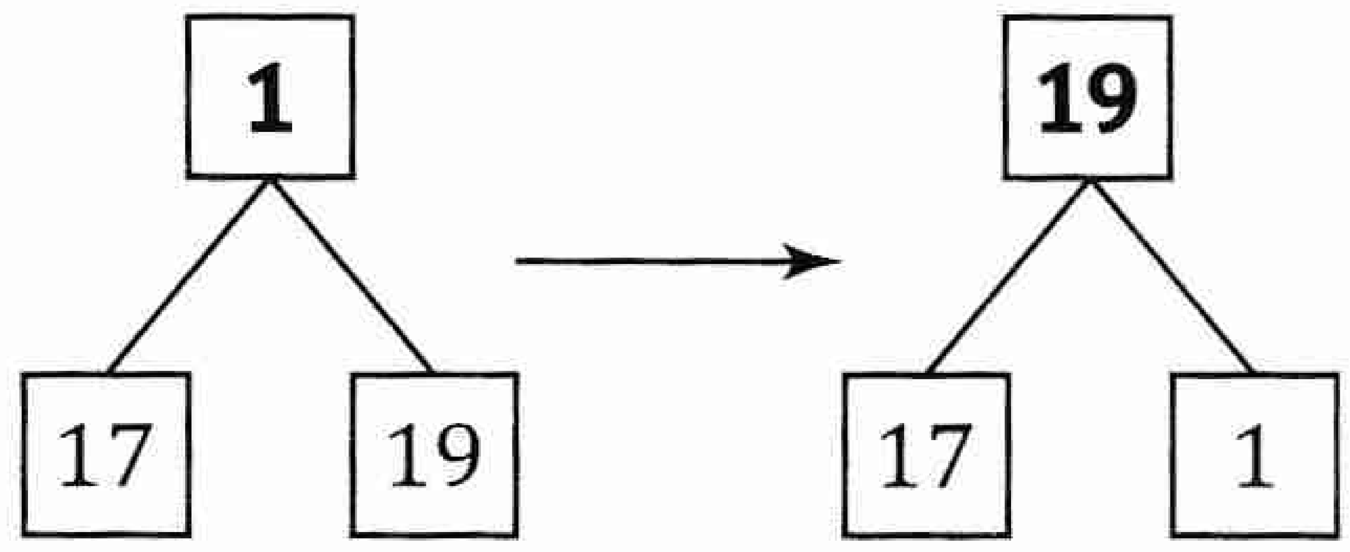
ছবি 9.10

নোডগুলোকে একটি অ্যারেতে রাখলে হবে এমন— heap = {0, 12, 7, 1, 3, 10, 17, 19, 2, 5}। অ্যারের 0 ইনডেক্সটি আমরা ব্যবহার করব না, তাই সেখানে 0 রেখেছি। এখন আমরা লুপ শুরু করব 4 নম্বর নোড (অর্থাৎ 3) থেকে। কারণ এর পরের নোডগুলোর কোনো চাইল্ড নেই। এই 4 সংখ্যাটি আমরা পেলাম heap_size অর্থাৎ 9-কে 2 দিয়ে ভাগ করে। max_heapify(heap, heap_size, 4) কল করলে নিচের সাবট্রি পরিবর্তিত হয়ে যাবে (ছবি 9.11)।



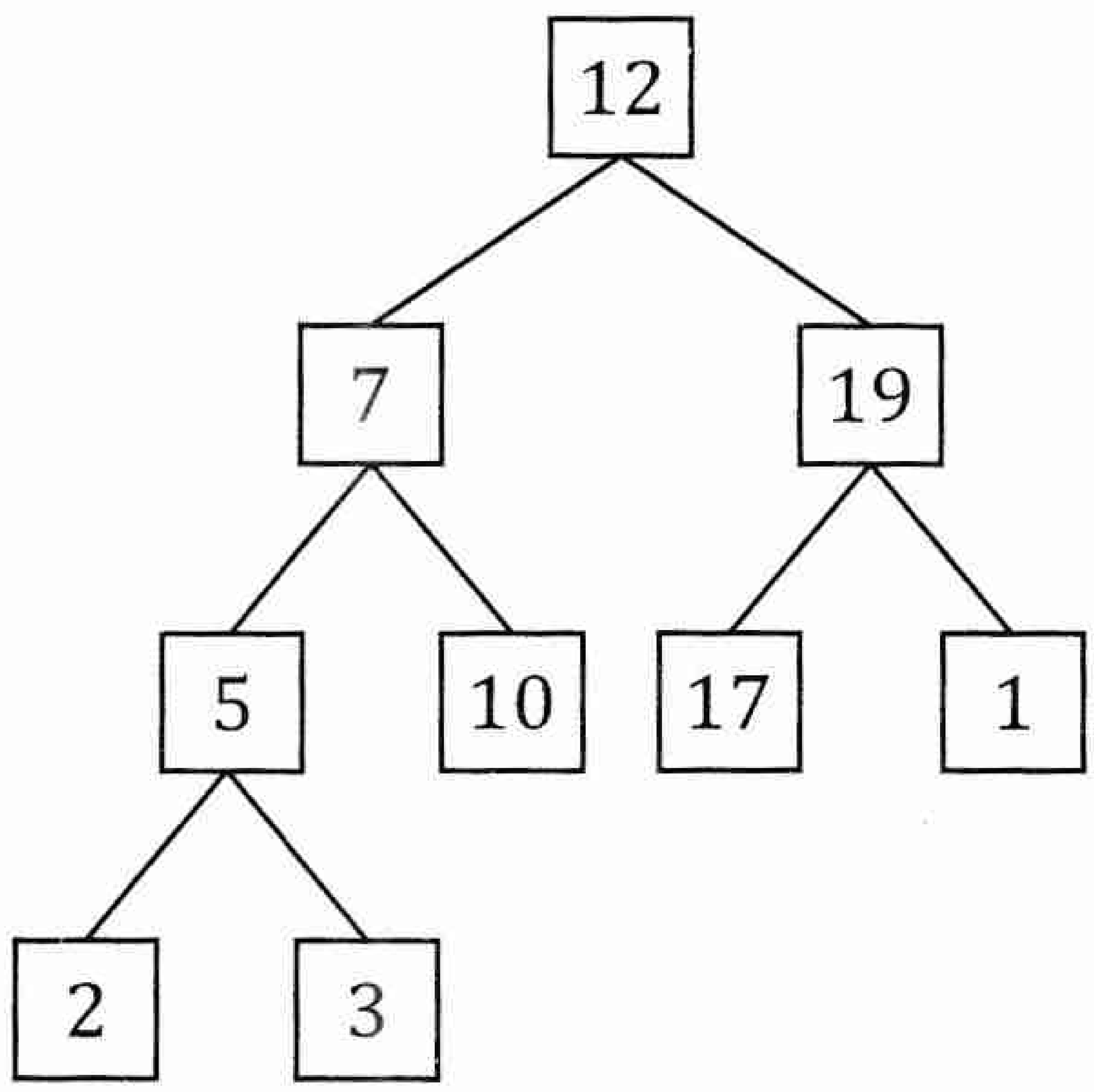
ছবি 9.11

তারপর max_heapify(heap, heap_size, 3) কল করা হবে। তখন ট্রি-তে ছবি 9.12-এর মতো পরিবর্তন ঘটবে।

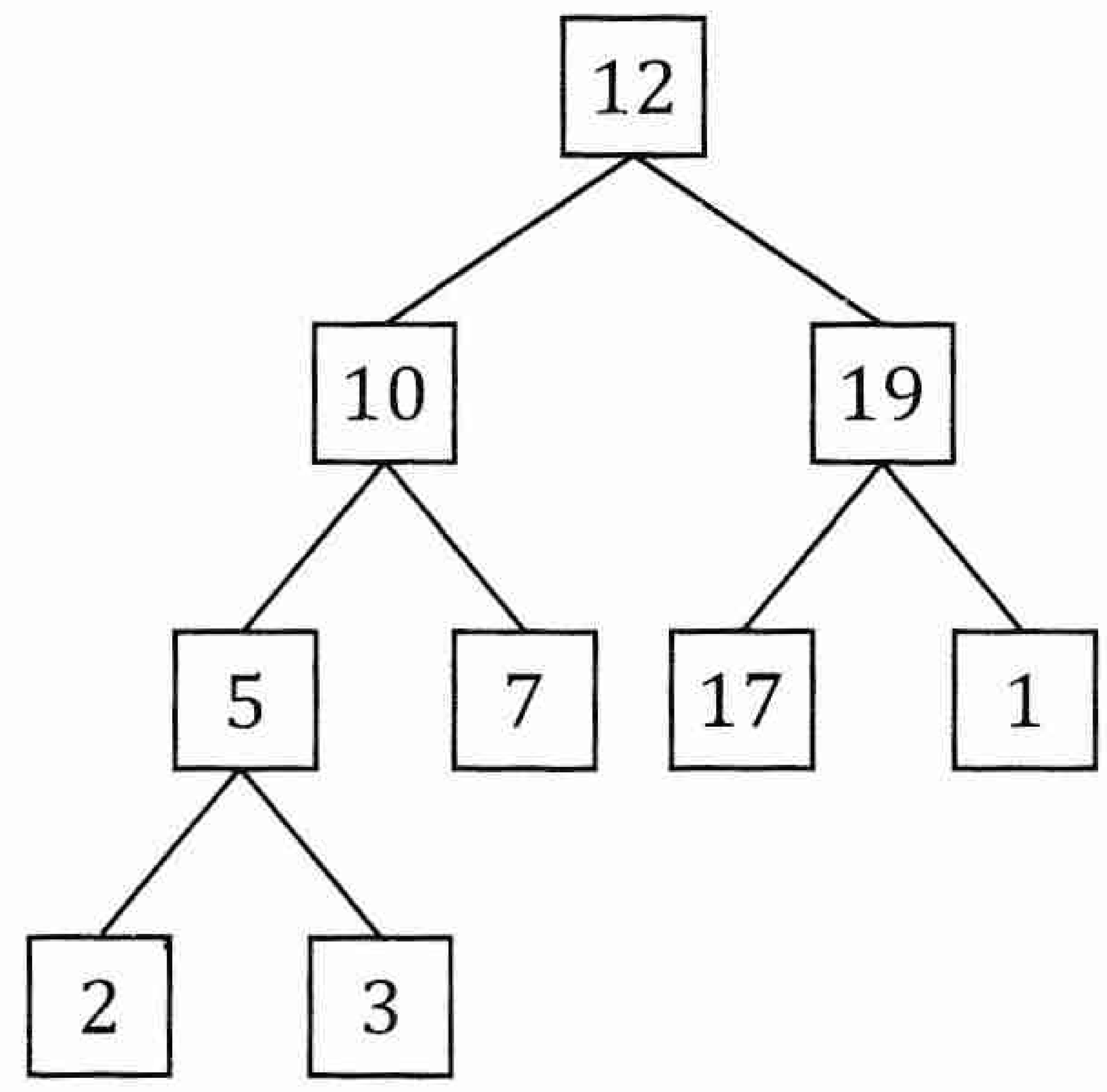


ছবি 9.12

এখন পুরো ট্রি-এর অবস্থা দাঁড়াবে ছবি 9.13-এর মতো–



ছবি 9.13



ছবি 9.14

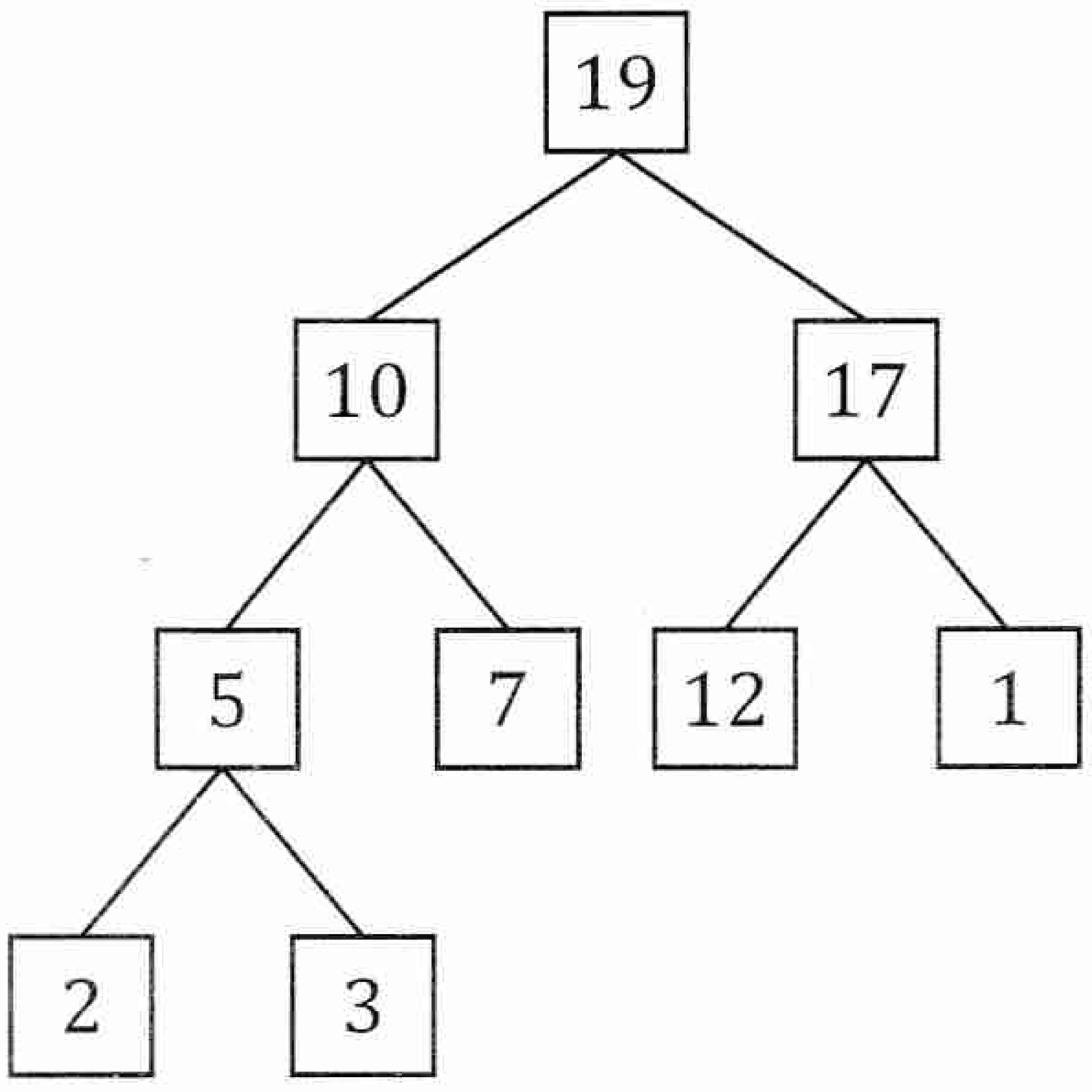
এখন max_heapify(heap, heap_size, 2) কল করা হবে। তাহলে ট্রি-তে পরিবর্তন ঘটে ছবি 9.14-এর মতো অবস্থা দাঁড়াবে।

এখন max_heapify(heap, heap_size, 1) কল করা হবে। তারপরে ট্রি-তে ছবি 9.15-এর মতো পরিবর্তন ঘটবে।

তাহলে আমাদের হিপ তৈরি হয়ে গেল। heap অ্যারেতে নোডগুলোর ক্রম হবে– 19, 10, 17, 5, 7, 12, 1, 2, 3। এখন তোমাদের কাজ হচ্ছে, মেইন ফাংশনে কোড লিখে build_max_heap() ফাংশনটি ঠিকভাবে কাজ করছে কি না, সেটি পরীক্ষা করা। কাজটি কিন্তু অবশ্যই করবে, নইলে শুধু বই পড়ে গেলে ঠিকমতো শেখা হবে না।

এখন, প্রশ্ন হচ্ছে max_heapify() এবং build_max_heap() ফাংশন দুটির কমপ্লেক্সিটি কত? max_heapify() ফাংশনের কমপ্লেক্সিটি হচ্ছে $O(\log n)$। কারণ কমপ্লিট বাইনারি ট্রি-তে nসংখ্যক নোড থাকলে তার হাইট (height) বা উচ্চতা হচ্ছে $\log n$। এটি কেন হয়, সেটি

চিন্তা করে বুঝতে হবে আর না বুঝলে বাইনারি সার্চের কথা মনে করবে। তার পরও না বুঝলে কোনো বন্ধুর সঙ্গে আলোচনা করতে হবে। আর build_max_heap() ফাংশনে আমরা একটি লুপ চালাই ($\frac{n}{2}$) আর সেই লুপের ভেতরে প্রতিবার max_heapify() ফাংশন কল করি। তাহলে build_max_heap() ফাংশনের কমপ্লেক্সিটি হচ্ছে $O(\frac{n}{2} \times \log n)$ বা $O(n \log n)$।



ছবি 9.15

হিপ সর্ট

এখন প্রশ্ন হচ্ছে, এই হিপ দিয়ে আমরা কী কাজ করব? হিপ ডেটা স্ট্রাকচার ব্যবহার করে সহজেই সর্টিং এর কাজ করা যায়। যেমন, আমরা জানি, ম্যাক্স হিপের ক্ষেত্রে হিপ তৈরি করার পরে সবচেয়ে বড়ো সংখ্যাটি থাকে রুট নোডে। তাই আমরা যেটি করতে পারি, সেটিকে লিস্টের শেষ উপাদানের সঙ্গে অদলবদল করতে পারি। তারপর শেষ নোড বাদে বাকি নোডগুলো নিয়ে যেই ট্রি থাকে, সেই ট্রি-এর রুট নোডের জন্য max_heapify() ফাংশন কল করতে পারি। হিপ সর্ট করার আগে কিন্তু হিপ তৈরি করে নিতে হবে– তোমরা নিশ্চয়ই জানো হিপ তৈরি করার জন্য কোন ফাংশন ব্যবহার করতে হবে।

```
void heap_sort(int heap[], int heap_size)
{
    int i, t;

    for (i = heap_size; i > 1; i--) {
        t = heap[1];
        heap[1] = heap[i];
        heap[i] = t;
```

```
        heap_size -= 1;
        max_heapify(heap, heap_size, 1);
    }
}
```

আমরা নতুন একটি সর্টিং অ্যালগরিদম শিখে ফেললাম, যার কমপ্লেক্সিটি হচ্ছে $O(n \log n)$। কারণ হিপসর্টের ভেতরে আমরা nসংখ্যক বার max_heapify() ফাংশন কল করছি, আর max_heapify-এর কমপ্লেক্সিটি হচ্ছে $O(\log n)$।

অনুশীলনী :

মেইন ফাংশনের ভেতরে একটি অ্যারে তৈরি করতে হবে। সেই অ্যারেকে ম্যাক্স হিপ বানাতে হবে। তারপর ম্যাক্স হিপকে হিপসর্ট করতে হবে। হিপসর্ট করার আগে ও পরে অ্যারেটি প্রিন্ট করে দেখতে হবে।

প্রায়োরিটি কিউ (Priority Queue)

আমরা ইতিমধ্যে কিউ নামে একটি ডেটা স্ট্রাকচারের সঙ্গে পরিচিত হয়েছি, যেখানে *আগে-আসলে-আগে-পাবেন* ভিত্তিতে কাজ করা হয়। অর্থাৎ, কিউ হচ্ছে সাধারণ লাইনের মতো, যে নতুন আসবে সে সবার পেছনে দাঁড়াবে আর লাইনে সবার আগে যে আছে, তাকে ডাকা হবে। কিন্তু মাঝেমধ্যে এমন হয় যে, গুরুত্বপূর্ণ কেউ এলে তাকে আগে জায়গা করে দেওয়া হয়। এখন, কিউ ডেটা স্ট্রাকচারে তো সেটি সম্ভব নয়, কারণ সে তো আর কারো গুরুত্ব বোঝে না। তবে কম্পিউটার বিজ্ঞানে এজাতীয় সমস্যার সমাধান আছে, যার নাম হচ্ছে প্রায়োরিটি কিউ। ইংরেজি প্রায়োরিটি (priority) শব্দের বাংলা অর্থ অগ্রাধিকার। প্রায়োরিটি কিউতে প্রতিটি নোডের সঙ্গে একটি মান যুক্ত থাকে। সেই মানের ভিত্তিতে অগ্রাধিকার নির্ধারণ করা হয়। প্রায়োরিটি কিউ দুই রকমের হতে পারে— ম্যাক্স (max) প্রায়োরিটি কিউ ও মিন (min) প্রায়োরিটি কিউ। আমরা যদি প্রায়োরিটি কিউতে সবচেয়ে বড়ো মানবিশিষ্ট নোডকে অগ্রাধিকার দিতে চাই, তাহলে ম্যাক্স প্রায়োরিটি কিউ তৈরি করব, আর যদি সবচেয়ে ছোটো মান বিশিষ্ট নোডকে অগ্রাধিকার দিতে চাই, তাহলে মিন প্রায়োরিটি কিউ তৈরি করব। আমরা এখানে ম্যাক্স প্রায়োরিটি কিউ নিয়ে আলোচনা করব। আর ম্যাক্স প্রয়োরিটি কিউ ঠিকভাবে বুঝলে মিন প্রায়োরিটি কিউ সহজেই তৈরি করা যাবে।

প্রায়োরিটি কিউ তৈরি করার সহজ উপায় হচ্ছে হিপ ডেটা স্ট্রাকচার ব্যবহার করা। আমরা জানি, ম্যাক্স হিপে সবচেয়ে বড়ো নোডটি সবার শুরুতে থাকে। তাই ম্যাক্স প্রায়োরিটি কিউ তৈরি করার জন্য আমরা ম্যাক্স হিপ ব্যবহার করব। প্রায়োরিটি কিউতে আমরা তিনটি জিনিস করতে চাই—

1. অগ্রাধিকার অনুযায়ী সবচেয়ে প্রথম নোডটি বের করা।
2. প্রায়োরিটি কিউতে নতুন নোড যুক্ত করা।
3. কোনো নোডের মান বৃদ্ধি করে তার প্রায়োরিটি বাড়িয়ে দেওয়া।

এখন আমরা যদি প্রয়োরিটি কিউ-এর সবচেয়ে বড়ো নোডটি পেতে চাই, তার জন্য হিপের প্রথম উপাদানটি নিতে পারলেই হবে। এর কমপ্লেক্সিটি হবে O(1)।

```
int get_maximum(int heap[])
{
    return heap[1];
}
```

আর আমরা যদি চাই যে, সবচেয়ে বড়ো নোডটি প্রয়োরিটি কিউ থেকে বের করে ফেলব, সেটিও করা যাবে। তবে, হিপের প্রথম বা রুট নোডটি আমরা যখন নিয়ে নেব, তখন সেই জায়গাটি খালি হয়ে যাবে। তাই সেই স্থানটি আমাদেরকে পূর্ণ করতে হবে। সেই কাজটি আমরা করতে পারি হিপের একেবারে শেষ নোডটি রুটে বসিয়ে দিয়ে। তারপরে সেই নোডের জন্য max_heapify() ফাংশন কল করব, যেন সেটি তার মান অনুযায়ী সঠিক স্থানে বসে যায়।

```
int extract_max(int heap[], int heap_size)
{
    int max_item = heap[1];

    // হিপের শেষ উপাদানটি প্রথম স্থানে (অর্থাৎ রুটে) কপি করি।
    heap[1] = heap[heap_size];

    // হিপের দৈর্ঘ্য এক কমে যাবে, কারণ শেষ উপাদানটি আমাদের আর দরকার নেই।
    heap_size -= 1;

    // শেষ উপাদানটি রুটে কপি করার পরে হিপ নষ্ট হয়ে যেতে পারে, যদি রুট তার
    // যেকোনো চাইল্ড-এর চেয়ে ছোটো হয়। তাই রুট নোডে (বা 1 ইনডেক্সে)
    // max_heapify() ফাংশন কল করতে হবে। তাহলে হিপটি ঠিক হয়ে যাবে
    // (heap[1]-এ যেই মানটি আছে, সেটি যথাস্থানে বসে যাবে)।
    max_heapify(heap, heap_size, 1);
```

```
    // এখন আমরা max_item রিটার্ন করতে পারি।
    return max_item;
}
```

এ ক্ষেত্রে যেহেতু max_heapify() ফাংশন একবার কল করা হচ্ছে, তাই extract_max() ফাংশনের কমপ্লেক্সিটি হবে $O(\log n)$। ওপরের দুটি ফাংশন (get_maximum ও extract_max) কল করার সময় একটি জিনিস খেয়াল রাখতে হবে। সেটি হচ্ছে, হিপে যেন কমপক্ষে একটি নোড থাকে, অর্থাৎ heap_size >= 1 হতে হবে।

এখন আমরা দেখব, কীভাবে প্রায়োরিটি কিউতে নতুন নোড যুক্ত করতে হয়। এজন্য আমরা হিপের শেষে নতুন নোডটি যুক্ত করব। তারপরে তার প্যারেন্ট নোডের সঙ্গে তুলনা করে দেখব যে, নোডটি তার প্যারেন্টের চেয়ে বড়ো কি না। যদি বড়ো না হয়, তাহলে তো আর কিছু করার দরকার নেই। আর যদি সে তার প্যারেন্টের চেয়ে বড়ো হয়, তাহলে প্যারেন্টের সঙ্গে জায়গা অদলবদল করতে হবে। তারপর আবার তার নতুন প্যারেন্টের সঙ্গে তাকে তুলনা করতে হবে। এরকমভাবে যতক্ষণ সে তার প্যারেন্টের চেয়ে বড়ো হচ্ছে, ততক্ষণ এই কাজ চালিয়ে যেতে হবে। প্যারেন্টের সমান বা প্যারেন্টের চেয়ে ছোটো হলে লুপ থেমে যাবে। আবার যদি দেখি এরকম করতে করতে সে ট্রি-এর রুটে পৌঁছে গেছে, তখনো আমরা কাজ থামিয়ে দেব।

```
int insert_node(int heap[], int heap_size, int node)
{
    int i, p, t;

    heap_size += 1;
    heap[heap_size] = node;

    // i > 1 শর্তটি মিথ্যা হবে, যখন i-এর মান 1 হবে (আমরা রুট নোডে পৌঁছে
    // গেলে i-এর মান 1 হবে), তাই তখন আর লুপ চলবে না। heap[i] যদি
    // heap[parent(i)]-এর সমান বা ছোটো হয়, তখন দ্বিতীয় শর্তটি
    // মিথ্যা হয়ে যাবে; তাই লুপও থেমে যাবে।
    while (i > 1 && heap[i] > heap[parent(i)]) {
        p = parent(i);

        t = heap[p];
        heap[p] = heap[i];
        heap[i] = t;

        i = p;
    }
```

```
    return heap_size;
}
```

insert_node() ফাংশনটির কমপ্লেক্সিটি হচ্ছে $O(\log n)$। কারণ নতুন যুক্ত নোডটি যদি শেষ পর্যন্ত রুটে গিয়ে পৌঁছায়, তাহলে $O(\log n)$সংখ্যক বার তার প্যারেন্টের সঙ্গে স্থান অদলবদল করতে হবে।

এখন কথা হচ্ছে, এই যে প্রায়োরিটি কিউ তৈরি করলাম, সেটি আমাদের কী কাজে লাগবে? বিভিন্ন অ্যালগরিদমে প্রয়োরিটি কিউ ব্যবহার করা হয়। কম্পিউটার অপারেটিং সিস্টেমেও কিছু কিছু কাজে প্রায়োরিটি কিউ-এর ব্যবহার রয়েছে। আবার আমরা যদি গ্রাহকসেবার একটি সফটওয়্যার তৈরি করি, যেখানে বিশেষ শ্রেণির গ্রাহকরা অগ্রাধিকারভিত্তিতে সেবা পাবে, সেখানেও প্রয়োরিটি কিউ ব্যবহার করা যেতে পারে। আর এ কথা বলাই বাহুল্য যে, প্রোগ্রামিং প্রতিযোগিতায় কিছু সমস্যার সমাধানে এটি ব্যবহার করা হয়। আবার অনেক আন্তর্জাতিক পর্যায়ের সফটওয়্যার নির্মাতা প্রতিষ্ঠানের নিয়োগের ইন্টারভিউতে প্রয়োরিটি কিউ-সংক্রান্ত সমস্যার সমাধান করতে হয়।

অনুশীলনী :

1. হিপের কোনো নোডের মান বৃদ্ধি করে তার প্রায়োরিটি বাড়িয়ে দেওয়ার ফাংশন লিখতে হবে। ফাংশনটির প্রোটোটাইপ হচ্ছে এরকম–

   ```
   void increase_key(int heap[], int i, int new_value);
   ```

   ফাংশনে heap[i]-তে new_value অ্যাসাইন করতে হবে। এই new_value কিন্তু নোডের বর্তমান মানের চেয়ে বড়ো হবে। তারপরে তার প্যারেন্টগুলোর সঙ্গে তুলনা করে নোডটিকে তার সঠিক স্থানে বসাতে হবে। এজন্য insert_node ফাংশনের লুপের মতোই কোড লিখতে হবে।
2. মিন প্রায়োরিটি কিউ-এর জন্য extract_min() ও insert_node() ফাংশন তৈরি করতে হবে।
3. একটি ম্যাক্স হিপ দেওয়া আছে, যার নোডগুলো হচ্ছে 19, 10, 17, 5, 7, 12, 1, 2, 3। এখানে নতুন একটি নোড যুক্ত করতে হবে, যার মান 18। নোড যুক্ত করলে কী কী ঘটনা ঘটবে সেটি ধাপে ধাপে দেখাতে হবে।

অধ্যায় ১০ : হ্যাশ টেবিল

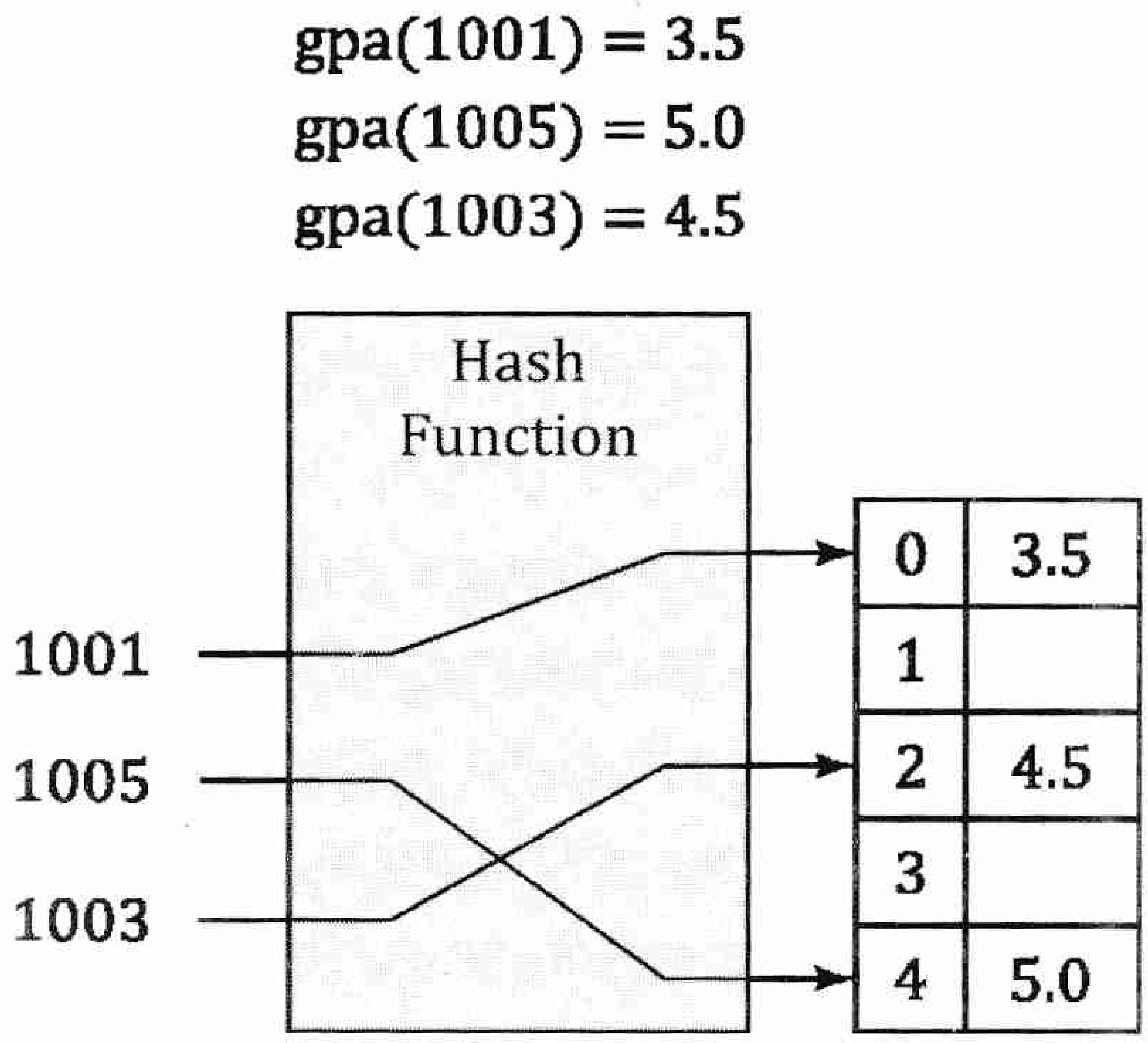
বইটি এতদূর পর্যন্ত যারা পড়েছ, তাই নিশ্চয়ই উপলব্ধি করেছ যে, একটি ডেটা স্ট্রাকচারে ডেটা রাখা, সেখান থেকে ডেটা নেওয়া, একটি নির্দিষ্ট ক্রমে ডেটাগুলো একসেস করা, কোনো একটি নির্দিষ্ট ডেটা একসেস করা– এই কাজগুলো মোটেও সহজ-সরল কোনো কাজ নয়। এই কাজগুলো করার জন্য অনেক চিন্তাভাবনা করে অনেক রকম পদ্ধতি বা অ্যালগরিদম আবিষ্কার করা হয়েছে। একেকটির আবার একেক রকম কমপ্লেক্সিটি। কিন্তু তোমাদের অনেকের মনেই হয়তো প্রশ্ন এসেছে, O(1) কমপ্লেক্সিটিতে যদি একটি ডেটা স্ট্রাকচারে ডেটা রাখা যেত এবং পড়া যেত, তাহলে কত ভালোই না হতো। এরকম ডেটা স্ট্রাকচার কিন্তু আছে, যার সঙ্গে তোমরা সবাই পরিচিত; এর নাম অ্যারে। কিন্তু অ্যারেতে তো ইনডেক্স হিসেবে কেবল পূর্ণসংখ্যা ব্যবহার করা যায়। অ্যারের ওপর ভিত্তি করে আমরা আরেক ধরনের ডেটা স্ট্রাকচার তৈরি করতে পারি, যার নাম হ্যাশ টেবিল– এটি কম্পিউটার বিজ্ঞানের একটি গুরুত্বপূর্ণ ধারণা। এই অধ্যায়ে আমরা হ্যাশ টেবিল সম্পর্কে প্রাথমিক জ্ঞান অর্জন করব।

ধরা যাক, একটি ক্লাসে 100 জন শিক্ষার্থী আছে। আমি যদি তাদের জিপিএ রাখতে চাই, তাহলে একটি অ্যারের মধ্যে তাদের জিপিএ রাখতে পারি। যদি অ্যারের নাম হয় gpa, তাহলে যার রোল নম্বর 1, তার জিপিএ থাকবে `gpa[0]`-এ, যার রোল নাম্বার 2 তার জিপিএ থাকবে `gpa[1]`-তে, এভাবে যার রোল নাম্বার 100, তার জিপিএ থাকবে `gpa[99]`-তে। আমরা যদি জানতে চাই, যার রোল নম্বর 60, তার জিপিএ কত, সেটি সহজেই পেয়ে যাব gpa[59]-এ। আবার কারো রোল নম্বর যদি 30 হয় এবং তার জিপিএ আমরা 4.3 দিতে চাই, সেটিও করা যাবে এভাবে : `gpa[29] = 4.3`। উভয় ক্ষেত্রেই (অর্থাৎ ডেটা নেওয়া ও রাখা) কমপ্লেক্সিটি হচ্ছে O(1)। কিন্তু এ আর নতুন কী? অ্যারে ব্যবহার করলে তো এরকমই হওয়ার কথা! এখন ধরা যাক, ক্লাসে শিক্ষার্থী 100 জন। কিন্তু তাদের রোল নম্বর হচ্ছে এরকম– 1001, 1002, 1003, ..., 1100। এখন তাদের জন্য কি আমি 1100 টি উপাদান বিশিষ্ট একটি অ্যারে (`double gpa[1100]`) তৈরি করব? চাইলে করতেই পারি, কিন্তু সেটি বেশি বুদ্ধিমানের কাজ হবে না, কারণ আমি অযথা প্রয়োজনের অতিরিক্ত মেমোরি খরচ করছি। আমি যা করতে পারি, তা হচ্ছে, একটি ছোট্ট ফাংশন তৈরি করে ফেলতে পারি, যা রোল নম্বরকে অ্যারের ইনডেক্সে রূপান্তর করবে। সেটি হতে পারে এরকম– `index = roll_number % 100 - 1`।

তাহলে,

যার রোল নম্বর 1001, তার জন্য `index` হবে, `1001 % 100 - 1 = 1 - 1 = 0`।

যার রোল নম্বর 1025, তার জন্য `index` হবে, `1025 % 100 - 1 = 25 - 1 = 24`।



ছবি 10.1

তাহলে আমরা লিস্টে 100টি উপাদান ব্যবহার করলেই হবে। আমরা এখানে যেই অ্যারেকে ব্যবহার করছি, তাকে চাইলে হ্যাশ টেবিল (hash table) বলতে পারি, আর যেই ফাংশনটি ব্যবহার করলাম, তাকে বলে হ্যাশ ফাংশন (hash function)। আমাদের হ্যাশ ফাংশনের উদাহরণটি বেশি সহজ, বাস্তবে হ্যাশ ফাংশন আরো জটিল হয়। যেমন, রোল নম্বর কিন্তু এরকমও থাকতে পারে– B1002084, C1012279। তখন আমরা হ্যাশ ফাংশন তৈরি করব কীভাবে? তখন আমাদের আরো চিন্তাভাবনা করতে হবে।

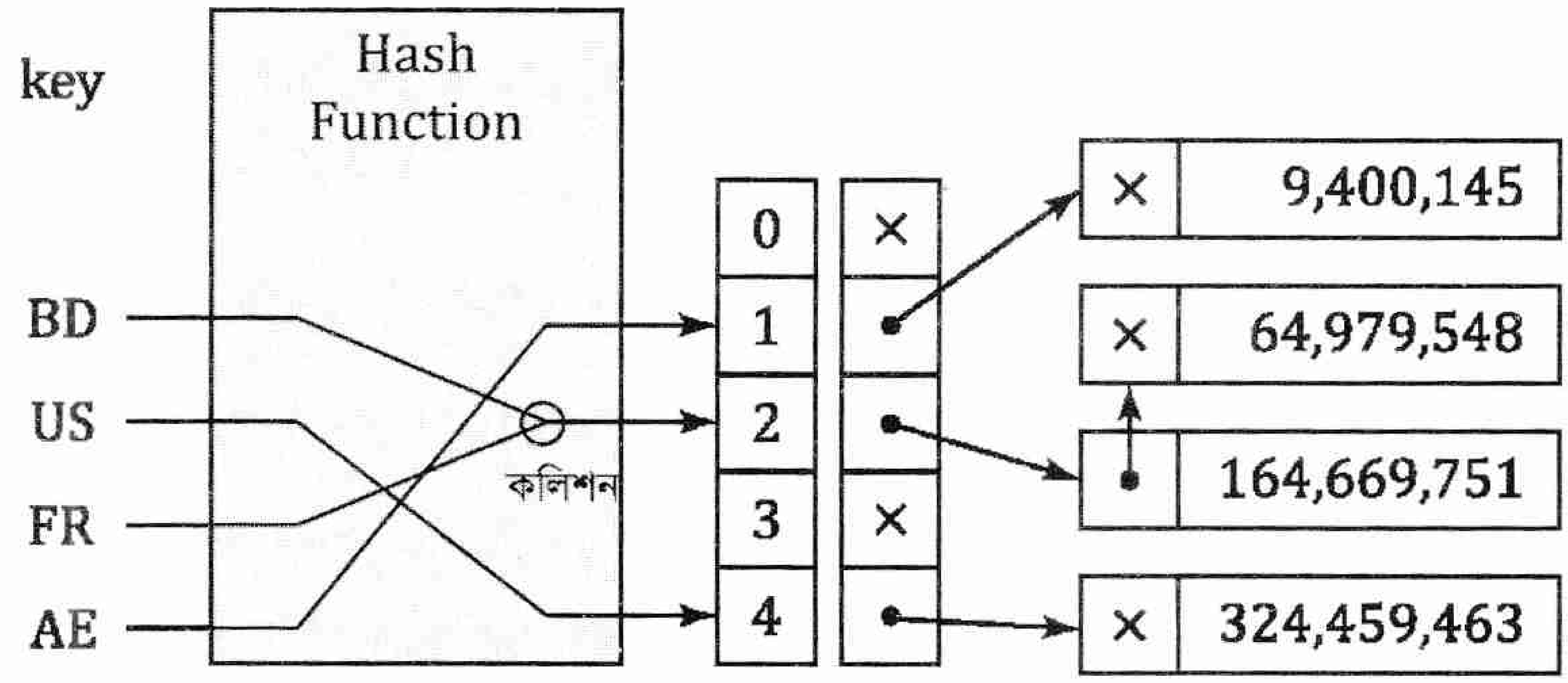
আবার, আমরা যদি আগে থেকে না জানি যে আসলে হ্যাশ টেবিলে মোট কতটি উপাদান থাকবে, তখন আমাদের অনুমান করে একটি সংখ্যা ব্যবহার করতে হবে। আন্দাজে যেকোনো সংখ্যা ব্যবহার করলেই হবে না, আমরা কেন হ্যাশ টেবিল তৈরি করতে চাই, সেটি বুঝে তারপরে অনুমান করা লাগবে। আমরা যদি একটি হ্যাশ টেবিল তৈরি করতে চাই, যেখানে বাংলাদেশের বিভিন্ন জেলার নাম হবে সেই টেবিলের কি (key), আর ভ্যালুতে বিভিন্ন তথ্য থাকবে, তখন আমরা বুঝব যে সেই টেবিলে আসলে 64-এর বেশি উপাদান দরকার নেই, কারণ বাংলাদেশে মোট 64টি জেলা আছে। কিন্তু যদি এমন হ্যাশ টেবিল তৈরি করতে চাই, যেখানে বাংলাদেশের মানুষের ন্যাশনাল আইডি থাকবে সেই টেবিলের কি (key), তখন আমাদের কিন্তু 50 বা 100 কোটি সাইজের হ্যাশ টেবিল তৈরি করা লাগবে (জনসংখ্যা তো 17 কোটি, তাহলে আমরা বেশি জায়গা নিচ্ছি কেন? উত্তর হচ্ছে জন্ম-মৃত্যু)।

যেহেতু হ্যাশ টেবিলের ভেতরে আমরা অ্যারে ব্যবহার করছি, তাই হ্যাশ টেবিলের কি (key) হিসেবে সংখ্যা, স্ট্রিং বা অন্য যা-কিছুই ব্যবহার করি না কেন, শেষ পর্যন্ত তাকে একটি সংখ্যায়

রূপান্তর করতে হবে। কারণ, অ্যারের ইনডেক্স হতে হয় একটি ধনাত্মক পূর্ণসংখ্যা (ইন্টিজার)। এই কাজটি করা হয় হ্যাশ ফাংশন ব্যবহার করে। সেখানে একটি ডেটা ইনপুট দিলে আউটপুট হিসেবে আমরা একটি সংখ্যা পাব যা ওই অ্যারের ইনডেক্স হিসেবে ব্যবহার করা যায়।

এখন হ্যাশ ফাংশনের আরেকটি উদাহরণ দেখি। ধরা যাক, আমি একটি হ্যাশ ফাংশন তৈরি করলাম, যে কিনা, ইনপুটের প্রতিটি অক্ষরের আসকি (ASCII) মান যোগ করে 100 দিয়ে মড (%) করবে। 100 দিয়ে মড করার কারণ হচ্ছে, আমি ধরে নিচ্ছি, আমি কোনোভাবে জানি যে, এই হ্যাশ টেবিলটিতে কখনোই 100টির বেশি উপাদান রাখা হবে না। এখন আমার ইনপুট যদি হয় একটি স্ট্রিং ABC, তাহলে তাদের আসকি মানগুলো যোগ করলে আমরা পাব, 65 + 66 + 67 বা, 198। একে 100 দিয়ে মড করলে পাব 98। এখন আরেকটি ইনপুট স্ট্রিং যদি CBA হয়, তাহলেও কিন্তু একই ঘটনা ঘটবে, অর্থাৎ, এর জন্যও আমার হ্যাশ ফাংশন (67 + 66 + 65) % 100 বা, 98 রিটার্ন করবে। তাহলে আমরা অ্যারের 98-তম ঘরে কাকে রাখব? ABC, নাকি CBA? আমরা তো একটু বিপদের পড়ে গেলাম। এই বিশেষ বিপদটিকে বলা হয় কলিশন (collision) যার বাংলা অর্থ হচ্ছে সংঘর্ষ। এই উদাহরণে হয়তো আমি আরো কিছু বুদ্ধি-শুদ্ধি প্রয়োগ করে এই বিশেষ বিপদ থেকে উদ্ধার পেতে পারি। কিন্তু বাস্তব ক্ষেত্রে যত চিন্তাভাবনা করেই হ্যাশ ফাংশন তৈরি করা হোক না কেন, কলিশনের সম্ভাবনা থেকেই যায়।

দেশ	কোড	জনসংখ্যা
বাংলাদেশ	BD	164,669,751
যুক্তরাষ্ট্র	US	324,459,463
ফ্রান্স	FR	64,979,548
আরব আমিরাত	AE	9,400,145



ছবি 10.2

একটি ভালো হ্যাশ ফাংশন তৈরি করে কলিশনের সম্ভাবনা কমানো যায়, কিন্তু একে পুরোপুরি এড়ানো যায় না। তাই কলিশন হলে সেটি সামলানোর জন্য কয়েকটি পদ্ধতি ব্যবহার করা হয়।

একটি পদ্ধতি হচ্ছে, হ্যাশ ফাংশন যেই মান রিটার্ন করল, সেখানে যদি ইতিমধ্যেই কোনো উপাদান থেকে থাকে, তাহলে পরবর্তী কোনো খালি ঘর ব্যবহার করা। এই পদ্ধতির নাম হচ্ছে ওপেন অ্যাড্রেসিং (open addressing)। এই ওপেন অ্যাড্রেসিং-ও আবার কয়েকভাবে করা যায়। একটি হচ্ছে ওই ঘরের পরের প্রতিটি ঘর পরীক্ষা করে প্রথমে যেই ঘরটি ফাঁকা পাওয়া যাবে, তাকে ব্যবহার করা। আরেকটি পদ্ধতি হচ্ছে কয়েকটি ঘর বাদ দিয়ে দিয়ে পরীক্ষা করা। ওপেন অ্যাড্রেসিংয়ের আরেকটি পদ্ধতি হচ্ছে, দ্বিতীয় একটি হ্যাশ ফাংশন ব্যবহার করা। আমরা এগুলোর বিস্তারিত আলোচনায় যাব না। আর ওপেন অ্যাড্রেসিং ছাড়াও কলিশন এড়ানোর আরেকটি পদ্ধতি আছে, যার নাম সেপারেট চেইনিং (seperate chaining)। সেই পদ্ধতিতে, অ্যারের প্রতিটি ঘরে একটি নির্দিষ্ট উপাদান রাখার বদলে সেখানে একটি লিংকড লিস্ট রাখা হয় এবং নতুন কোনো উপাদান সেই ঘরে আসলে তাকে ওই লিংকড লিস্টের শেষে রাখা হয়। আর এই কলিশন হওয়ার সম্ভাবনা থাকে বলেই, হ্যাশ টেবিলের ডেটা রাখা বা পড়ার কমপ্লেক্সিটি যে $O(1)$ হবেই, এর নিশ্চয়তা দেওয়া যায় না।

হ্যাশ টেবিল ও হ্যাশ ফাংশনের ধারণাটি বেশ সহজ। কিন্তু বিষয়টি ইমপ্লিমেন্ট করা মোটেও সহজ কোনো কাজ হয়। ভালো হ্যাশ ফাংশন তৈরি করতে প্রচুর গবেষণা প্রয়োজন। বিষয়টি সম্পর্কে আরো জানতে চাইলে উইকিপিডিয়া থেকে পড়তে হবে। এ ছাড়া থমাস কোরম্যানের *Introduction to Algorithms* বইতে হ্যাশ টেবিল নিয়ে বিস্তারিত আলোচনা আছে।

অধ্যায় ১১ : আরো সর্টিং অ্যালগরিদম

মার্জ সর্ট (Merge sort)

ইংরেজি শব্দ মার্জ (merge)-এর অর্থ হচ্ছে একত্র করা। যেমন, অনেক সময় আমরা বলি, দুটি কোম্পানি মার্জ হয়েছে, মানে দুটি প্রতিষ্ঠান একত্রিত হয়ে একটি প্রতিষ্ঠানে পরিণত হয়েছে। মার্জ সর্ট অ্যালগরিদমের মূলমন্ত্র হচ্ছে, যেই অ্যারেকে সর্ট করতে হবে, তাকে ভেঙে দুটি অ্যারে বানিয়ে ফেলা, তারপরে সেই আলাদা দুটি অ্যারেকে সর্ট করা। অবশেষে সর্টকৃত অ্যারে দুটিকে একত্র করা বা মার্জ করা, যেন তারা মার্জ করে যখন একটি অ্যারে হবে, তখনো যেন তারা সর্ট করা অবস্থায় থাকে।

তাহলে মার্জ সর্ট অ্যালগরিদমটি আমরা লিখে ফেলি—

মার্জ সর্ট অ্যালগরিদম

ইনপুট : একটি অ্যারে A।

ধাপ 1 : অ্যারেতে যদি একটি বা তার চেয়ে কম উপাদান থাকে, তাহলে সেই অ্যারেটি ইতিমধ্যে সর্ট করা।

ধাপ 2 : A অ্যারের প্রথম অর্ধেক উপাদান left নামক অ্যারেতে রাখি। left অ্যারেকে মার্জ সর্ট করি।

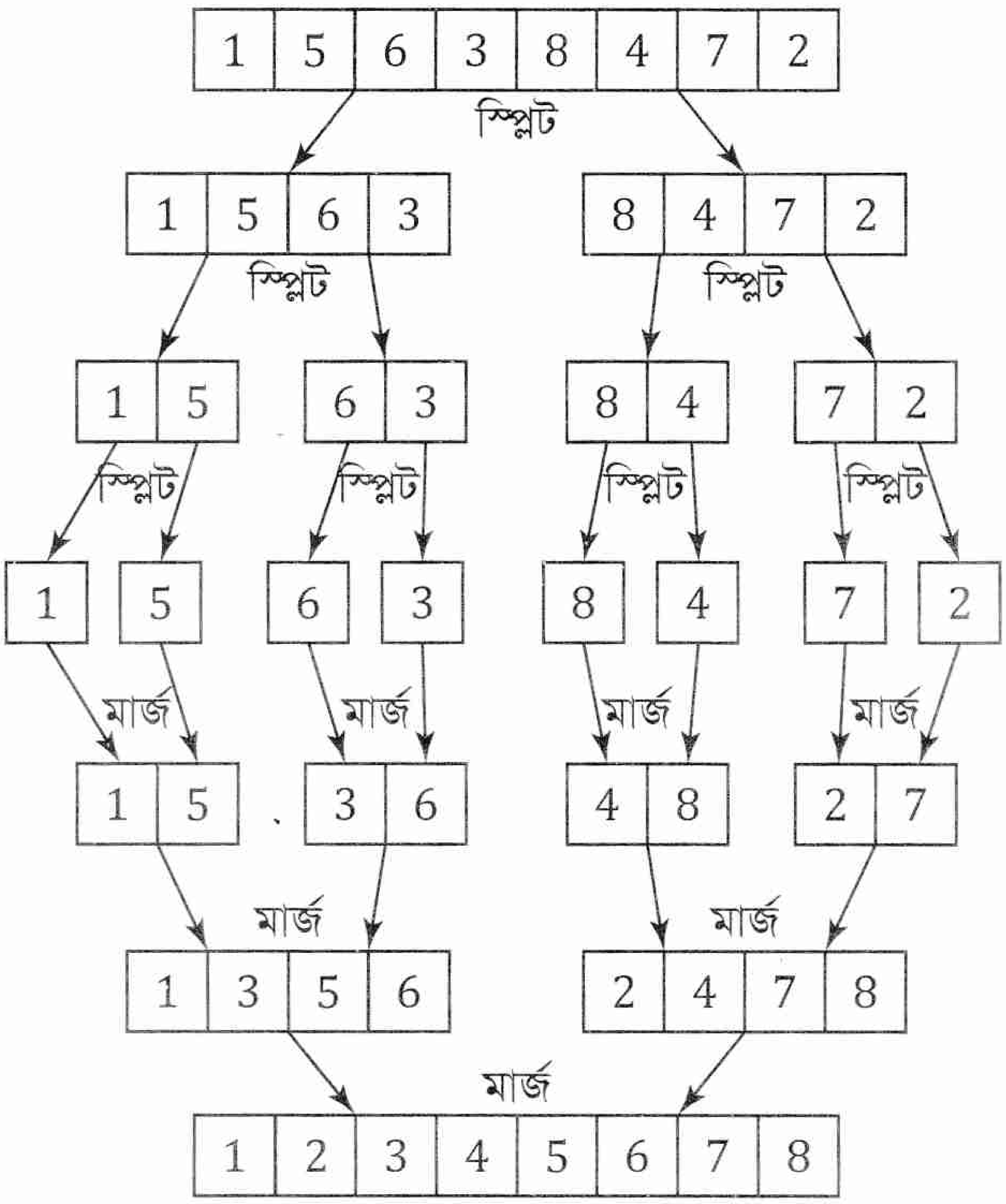
ধাপ 3 : A অ্যারের দ্বিতীয় অর্ধেক উপাদান right নামক অ্যারেতে রাখি। right অ্যারেকে মার্জ সর্ট করি।

ধাপ 4 : left অ্যারে ও right অ্যারে মার্জ করি।

ছবি 11.1-এ একটি পূর্ণসংখ্যার অ্যারেকে মার্জ সর্ট করার পদ্ধতি ধাপে ধাপে দেখানো হলো।

আমরা ছবি 11.1-এ দেখতে পাচ্ছি যে, যতক্ষণ অ্যারেতে একাধিক সংখ্যা আছে, আমরা সেই অ্যারেকে ভেঙে ফেলছি (ইংরেজিতে বলে split করা) এবং যখন প্রতি অ্যারেতে একটি উপাদান থাকছে, তার পরের ধাপ থেকে আমরা সেগুলো মার্জ করছি। আশা করি, মার্জ সর্ট পদ্ধতিটি কারো বুঝতে সমস্যা হবে না। তার পরও ইন্টারনেটে মার্জ সর্টের প্রচুর ছবি ও অ্যানিমেশন আছে, সেগুলো দেখে নেওয়া যেতে পারে। এই ধরনের অ্যালগরিদমকে বলে ডিভাইড অ্যান্ড কনকোয়ার (divide and conquer)। কারণ আমরা সমস্যাকে ছোটো ছোটো ভাগে ভাগ করছি (divide),

তারপর ছোটো সমস্যার সমাধান করতে করতে বড়ো সমস্যাটির সমাধান করে ফেলছি (conquer)।



ছবি 11.1

এখন আমরা মার্জ সর্ট ইমপ্লিমেন্ট করব। আমরা ইতিমধ্যেই জেনেছি যে, যেই অ্যারেকে সর্ট করতে হবে, সেটিকে দুই ভাগ করে ফেলতে হবে। তারপর প্রতি ভাগকে আলাদাভাবে সর্ট করে মার্জ করতে হবে। আমরা কিন্তু রিকার্শনের গন্ধ পাচ্ছি। কিন্তু মার্জ সর্টের বেইজ কন্ডিশন কী হবে? ফ্যাক্টরিয়াল বের করার সময় যেমন আমরা জানি যে, $1! = 1$। আবার ফিবোনাচ্চি সংখ্যা বের করার সময়ও আমরা প্রথম দুটি ফিবোনাচ্চি সংখ্যার মান জানি। মার্জ সর্টের বেলাতে আমরা বলতে পারি, একটি অ্যারেতে যদি একটিমাত্র উপাদান থাকে, তাহলে সেই অ্যারে ইতিমধ্যে সর্ট করা আছে, তা সে যে ক্রমেই (ছোটো থেকে বড়ো কিংবা বড়ো থেকে ছোটো) সাজাতে বলা হোক না কেন। আবার অ্যারেতে যদি কোনো উপাদান না থাকে, তাহলে সেই অ্যারেও সর্ট করা আছে (এটি একটু অদ্ভুত শোনাতে পারে যে, অ্যারেতে কিছু না থাকলে সেটি আবার সর্ট করা থাকে কীভাবে, কিন্তু একটু সূক্ষ্মভাবে চিন্তা করলে বোঝা যাবে, ব্যাপারটি কিন্তু সত্যি)।

মার্জ সর্ট করার জন্য আমরা একটি ফাংশন লিখে ফেলতে পারি—

```
// অ্যারে A-কে মার্জ সর্ট করতে হবে।
// left-হচ্ছে অ্যারের প্রথম উপাদানের ইনডেক্স।
// right-হচ্ছে অ্যারের শেষ উপাদানের ইনডেক্স।

void merge_sort(int A[], int left, int right)
{
    if (left >= right) {
        // ওপরের শর্ত সত্য হওয়ার মানে হচ্ছে অ্যারেতে একটি (left ও right
        // সমান হলে) বা শূন্যটি (left > right হলে) উপাদান আছে।
        // তার মানে, আমরা বলতে পারি এটি সর্ট করাই আছে।
        // তাই ফাংশন থেকে রিটার্ন করব।
        return;
    }

    // অ্যারেকে যেহেতু দুই ভাগে ভাগ করতে হবে, তাই আমরা মাঝখানের ইনডেক্সটি
    // বের করব।
    // অ্যারের প্রথমভাগ হবে left থেকে mid পর্যন্ত,
    // দ্বিতীয়ভাগ হবে
    // mid+1 থেকে right পর্যন্ত।
    int mid = left + (right - left) / 2;

    // A-এর প্রথম অংশ (A[left] থেকে A[mid] পর্যন্ত) মার্জ সর্ট করতে হবে।
    merge_sort(A, left, mid);

    // A-এর দ্বিতীয় অংশ (A[mid+1] থেকে A[right] পর্যন্ত) মার্জ সর্ট
    // করতে হবে।
    merge_sort(A, mid+1, right);

    // প্রথম অ্যারে (A[left] থেকে A[mid]) ও দ্বিতীয় অ্যারে (A[mid+1]
    // থেকে A[right]) মার্জ করতে হবে। আমরা আলাদা অ্যারে তৈরি না করে,
    // বরং left, mid, right ইনডেক্স দিয়ে A-অ্যারেকে দুটি অংশে ভাগ করছি।
    merge(A, left, mid, right);
}
```

এখন আমরা দেখে নিই, দুটি সর্টেড অ্যারে আমরা কীভাবে মার্জ করতে পারি। মার্জ করার জন্য আমরা একটি ফাংশন লিখব, যার প্রোটোটাইপ হবে এমন—

```
void merge(int A[], int left, int mid, int right)
```

দুটি সর্টেড অ্যারে মার্জ করা মোটেও কঠিন কোনো ব্যাপার নয়। তোমরা যারা *কম্পিউটার প্রোগ্রামিং ১ম খণ্ড ও ২য় খণ্ড* বই দুটো ভালোভাবে পড়েছ এবং চর্চা করেছ, তারা দুই ঘণ্টা চেষ্টা করলে নিজেরাই কাজটি করে ফেলতে পারবে। তোমাদের উচিত হবে বইটি বন্ধ করে নিজে নিজে মার্জ ফাংশনটি তৈরি করার চেষ্টা করা। তারপরে আবার বই পড়া শুরু করবে।

আমরা merge ফাংশনের ভেতরে L ও R নামে দুটি অ্যারে ব্যবহার করব। A[left] থেকে A[mid] পর্যন্ত উপাদানগুলো L অ্যারেতে কপি করব, আর A[mid+1] থেকে A[right] পর্যন্ত উপাদানগুলো R অ্যারেতে কপি করব। তারপর মার্জ করার জন্য একটি লুপ চালাতে হবে। সেখানে L ও R-এর প্রথম দুটি উপাদান পরস্পর তুলনা করে যেটি ছোটো, সেটি A-তে রাখব। তারপরে ছোটো উপাদানটি যেই অ্যারে থেকে নেওয়া হয়েছিল, সেই অ্যারের পরবর্তী উপাদানের সঙ্গে অন্য অ্যারের প্রথম উপাদানটি তুলনা করব। এভাবে চলতে থাকবে। প্রতিবার যেই অ্যারে থেকে একটি উপাদান নেব, পরেরবারে সেই অ্যারের পরের উপাদানে চলে যাব ইনডেক্সের মান এক বাড়িয়ে দিয়ে। লুপটি কতবার চলবে? যতক্ষণ না কোনো একটি অ্যারের ইনডেক্স তার দৈর্ঘ্যের সমান হয়ে যাচ্ছে, ততক্ষণ লুপ চলবে। তোমরা নিশ্চয়ই ফাংশনটির কোড দেখার জন্য অস্থির হয়ে গিয়েছ। আমি কোড নিচে লিখে দিলাম।

```
void merge(int A[], int left, int mid, int right)
{
    int i;
    int index_a, index_l, index_r;
    int size_left, size_right;

    size_left = mid - left + 1;
    size_right = right - mid;

    int L[size_left], R[size_right];

    // A[left] থেকে A[mid] পর্যন্ত সংখ্যাগুলো L অ্যারেতে কপি করি।
    for (i = 0; i < size_left; i++) {
        L[i] = A[left + i];
    }

    // A[mid+1] থেকে A[right] পর্যন্ত সংখ্যাগুলো R অ্যারেতে কপি করি।
    for (i = 0; i < size_right; i++) {
        R[i] = A[mid + 1 + i];
```

```
}

index_l = 0;
index_r = 0;

for (index_a = left;
      index_l < size_left && index_r < size_right;
       index_a++) {

    if (L[index_l] < R[index_r]) {
    /*  যেহেতু L[index_l] ছোটো, তাই একে আমরা A[index_a]-তে
        রাখছি। তারপর index_l-এর মান এক বাড়িয়ে দিচ্ছি, যেন পরের বার
        L-এর পরবর্তী উপাদানের সঙ্গে R-এর উপদানের তুলনা করা হয়।
    */
        A[index_a] = L[index_l];
        index_l += 1;

    } else {
    /*  যেহেতু R[index_r], L[index_l]-এর চেয়ে ছোটো বা সমান,
        তাই R[index_r]-কে আমরা A[index_a]-তে রাখছি। তারপর
        index_r-এর মান এক বাড়িয়ে দিচ্ছি, যেন পরের বার L-এর
        উপাদানের  সঙ্গে R-এর পরবর্তী উপদানের তুলনা করা হয়।
    */
        A[index_a] = R[index_r];
        index_r += 1;
    }
}

// এই পর্যায়ে আসার পরে, হয় L  ও R-এর সব উপাদান মার্জ হয়ে A-তে চলে
// এসেছে। অথবা, L  ও R-এর  মধ্যে কোনো একটি অ্যারের কিছু উপাদান এখনো
// A-তে আসেনি। তাই নিচের দুটি লুপের  যেকোনো একটি চলবে, কখনো দুটি
// একসঙ্গে চলবে না। কেন? সেটি চিন্তা করো।

// L-এর কিছু উপাদান যদি অবশিষ্ট থাকে, সেগুলো A-তে কপি করি।
while (index_l < size_left) {
    A[index_a] = L[index_l];
    index_l += 1;
    index_a += 1;
}
```

```
    // R-এর কিছু উপাদান যদি অবশিষ্ট থাকে, সেগুলো A-তে কপি করি।
    while (index_r < size_right) {
        A[index_a] = R[index_r];
        index_r += 1;
        index_a += 1;
    }

    // এই জায়গায় এসে অ্যারে দুটি মার্জ হয়ে গিয়েছে,এবং মার্জ করার পর উপাদান-
    // গুলো A-তেই আছে। আমরা যেহেতু ফাংশনের প্যারামিটার হিসেবে A অ্যারে
    // পাঠিয়েছি, তাই সেটির ভেতরে পরিবর্তন হয়ে যাবে, আলাদাভাবে রিটার্ন করার
    // কিছু নেই।
}
```

এখন তোমরা একটি মেইন ফাংশন লিখে মার্জ সর্ট করে দেখতে পারো ঠিকঠাক সর্ট হচ্ছে কি না।

```
int main()
{
    int i, n = 8;
    int A[] = {1, 5, 6, 3, 5, 8, 7, 2, 9};

    merge_sort(A, 0, n);

    for (i = 0; i <= n; i++) {
        printf("%d  ", A[i]);
    }

    printf("\n");
}
```

তোমরা ওপরের প্রোগ্রামটি রান করে দেখো।

মার্জ সর্টের কমপ্লেক্সিটি কত? প্রতিবার আমরা অ্যারেকে দুই ভাগে ভাগ করে ফেলছি। তাই এই অংশের কমপ্লেক্সিটি হচ্ছে $O(\log n)$। কিন্তু আমরা আবার প্রতিবার অ্যারেকে ভাগ করার শেষে, জোড়াও লাগাচ্ছি। সেই জোড়া লাগানোর কাজটি, অর্থাৎ `merge()` ফাংশনের কমপ্লেক্সিটি হচ্ছে $O(n)$। তাই মার্জ সর্টের কমপ্লেক্সিটি হচ্ছে $O((\log n) \times n)$, বা $O(n \times \log n)$, যাকে আমরা সংক্ষেপে লিখি $O(n \log n)$।

কুইক সর্ট (Quick Sort)

বিভিন্ন প্রোগ্রামিং ভাষায় সর্ট করার জন্য বিল্ট-ইন ফাংশন তৈরি করে দেওয়া থাকে। সাধারণত এসব সর্ট ফাংশন তৈরির জন্য যেই অ্যালগরিদমটি ব্যবহার করা হয়, তার নাম হচ্ছে কুইক সর্ট। সবচেয়ে মজার ব্যাপার হচ্ছে, হিপ সর্ট ও মার্জ সর্ট অ্যালগরিদমের এভারেজ কেস ও ওয়ার্স্ট কেস রানটাইম দুটোই $O(n \log n)$ হলেও কুইক সর্টের এভারেজ কেস হচ্ছে $O(n \log n)$ আর ওয়ার্স্ট কেস রানটাইম হচ্ছে $O(n^2)$। তার পরও সঠিকভাবে ইমপ্লিমেন্ট করতে পারলে এটি হিপ সর্ট ও মার্জ সর্টের চেয়ে দ্রুতগতিতে ফলাফল দেয়। আর কেবল অ্যারে যদি আগে থেকেই সর্ট করা থাকে, তখন এর পারফরম্যান্স খারাপ হয়। যা-ই হোক, এখন আমরা কুইক সর্ট অ্যালগরিদম শিখব এবং সেটি ইমপ্লিমেন্ট করব।

মার্জ সর্টের মতো, কুইক সর্টও ডিভাইড অ্যান্ড কনকোয়ার ধরনের অ্যালগরিদম। এর মূলনীতি হচ্ছে, অ্যারে থেকে একটি সংখ্যা নির্বাচন করা, যাকে বলা হয় পিভোট (pivot) এবং সেই সংখ্যাকে অ্যারেতে তার সঠিক স্থানে এমনভাবে বসানো, যেন তার বাঁ দিকের সংখ্যাগুলো সেই সংখ্যার চেয়ে ছোটো আর ডান দিকের সংখ্যাগুলো সেই সংখ্যার চেয়ে বড়ো (কিংবা সমান) হয়—ছোটো থেকে বড়ো ক্রমে সাজানোর ক্ষেত্রে এমনটি করা হয়। এই কাজটি করার নাম হচ্ছে পার্টিশন করা। এখন প্রথমবার পার্টিশন করলে পিভোট সংখ্যাটি তার যথাস্থানে চলে আসবে। এরপরের কাজ হচ্ছে তার বাঁ দিকের সংখ্যাগুলোকে সর্ট করে ফেলা এবং ডান দিকের সংখ্যাগুলোকেও সর্ট করে ফেলা। অর্থাৎ আমরা এখন মূল অ্যারেকে ছোটো দুটি অ্যারেতে ভেঙে ফেললাম। মার্জ সর্টে যেমন সব সময় সমানভাবে ভাগ করতাম, কুইক সর্টের বেলাতে কিন্তু আমরা দুটি অ্যারেকে সমানভাগে ভাগ করি না। পিভোট-এর চেয়ে ছোটোগুলো একভাগে, আর তার চেয়ে বড়োগুলো (সমানও থাকতে পারে) একভাগে ভাগ করে আবার সর্ট করি। এজন্য পিভোট নির্বাচন খুব গুরুত্বপূর্ণ। আমরা যদি এরকম নিশ্চিত করতে পারি যে, ভাগগুলো প্রায় সমান-সমান হচ্ছে, তাহলে পারফরম্যান্স হবে $O(\log n)$। আর পার্টিশন করার জন্য আমাদের প্রয়োজন $O(n)$। তাই দুটি কাজ একসঙ্গে হয়ে $O(n \log n)$ সময় লাগে। তো এই পিভোট নির্বাচন করার সঠিক বা সর্বোত্তম পদ্ধতি নিয়ে বিস্তর গবেষণা হয়েছে। আমরা এই বইতে সেই আলোচনায় যাব না। বরং আমরা লিস্টের সবচেয়ে শেষ সংখ্যাটিকে পিভোট হিসেবে ধরব।

এখন আমরা তো অ্যারেকে দুটি ভাগে ভাগ করছি এবং সেগুলোকে আবার সর্ট করছি। আমরা নিশ্চয়ই বুঝতে পারছি, এখানেও আমাদেরকে রিকার্শন ব্যবহার করতে হবে। কিন্তু আমরা থামব কখন? একটি অ্যারেতে যদি এক বা তার চেয়ে কমসংখ্যক উপাদান থাকে, তাহলে আমরা বলতে পারব যে, সেই অ্যারে সর্ট করা আছে।

তাহলে কুইক সর্ট অ্যালগরিদমটি আমরা লিখে ফেলি।

কুইক সর্ট অ্যালগরিদম

ইনপুট : একটি অ্যারে, অ্যারের প্রথম উপাদানের ইনডেক্স (low) আর শেষ উপাদানের ইনডেক্স (high)।

ধাপ 1 : অ্যারেতে যদি এক বা তার চেয়ে কমসংখ্যক উপাদান থাকে, তাহলে রিটার্ন করি।

ধাপ 2 : অ্যারেকে এমনভাবে পার্টিশন করি, যেন অ্যারের শেষ উপাদানটি তার যথাস্থানে বসে যায়। ধরা যাক, সেই স্থানের মান, অর্থাৎ, ইনডেক্স হচ্ছে `p`।

ধাপ 3 : অ্যারের `low` থেকে `p-1` পর্যন্ত ইনডেক্স নিয়ে যেই অংশটুকু আছে, তাকে কুইক সর্ট করি।

ধাপ 4 : অ্যারের `p+1` থেকে `high` পর্যন্ত ইনডেক্স নিয়ে যেই অংশটুকু আছে, তাকে কুইক সর্ট করি।

ছবি 11.2-এ কুইক সর্ট কীভাবে কাজ করে, সেটি দেখানো হলো। এখানে অ্যারের সবচেয়ে শেষ উপাদানটি পিভোট হিসেবে ব্যবহার করা হয়েছে।

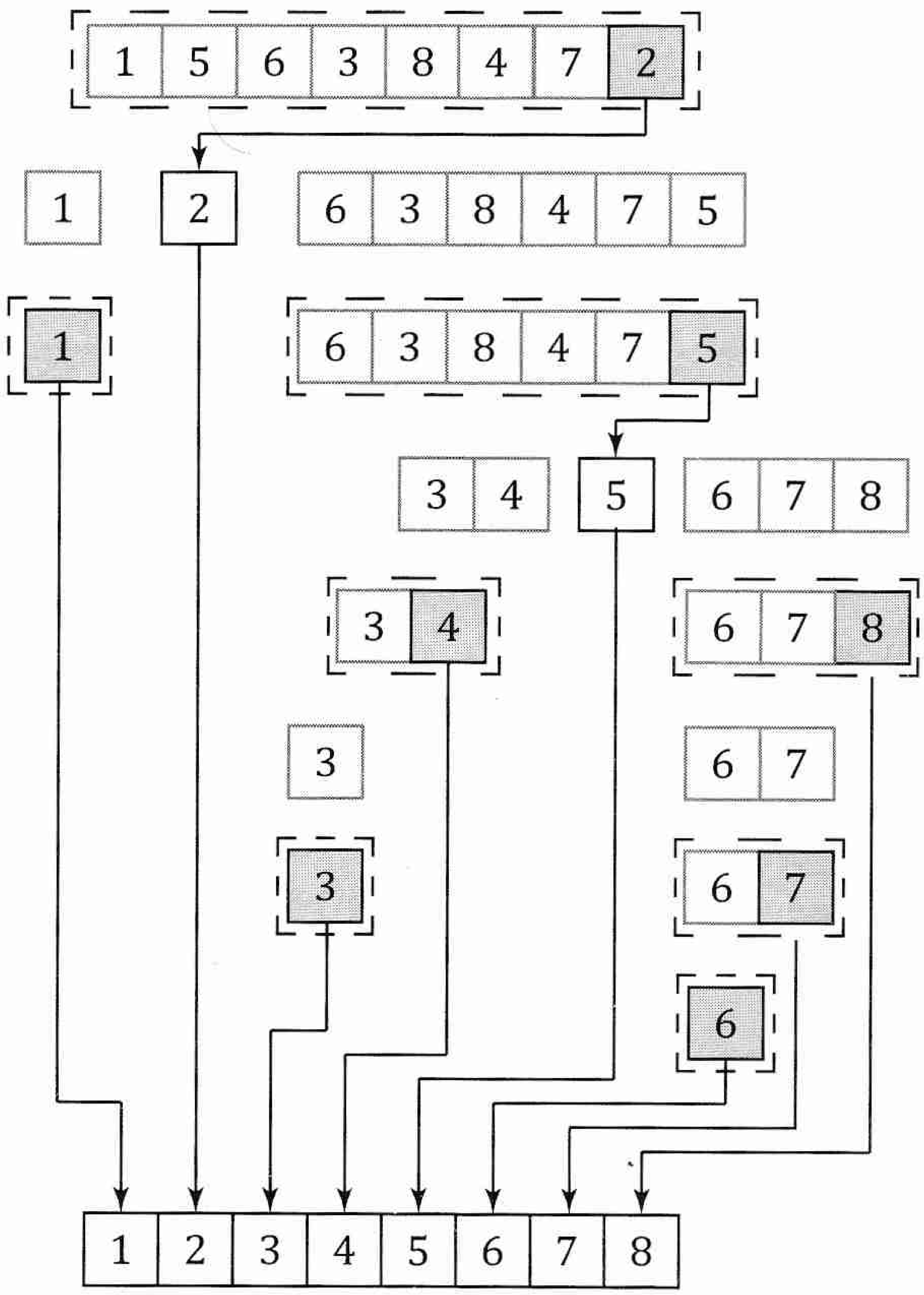
ধাপ 1 : প্রথমে অ্যারের সবচেয়ে শেষ উপাদানটি, অর্থাৎ 2-কে নিয়ে তার যথাস্থানে রাখলাম। এজন্য পার্টিশন পদ্ধতিতে 2-এর সঙ্গে 5-এর জায়গার অদলবদল হয়েছে। এটি কীভাবে হলো, সেটি আমরা একটু পরেই দেখব। এখন 2-এর বাঁ দিকের সবগুলো সংখ্যা 2-এর ছোটো, আর ডান দিকের সংখ্যাগুলো 2-এর চেয়ে বড়ো।

ধাপ 2 : 2-এর বাঁ দিকের অ্যারেকে এখন কুইক সর্ট করতে হবে। সেখানে মাত্র একটি উপাদান, তাই সেটি সর্ট করাই আছে। তারপর 2-এর ডান দিকের অ্যারেকে কুইক সর্ট করতে হবে। সেখানে পিভোট হচ্ছে 5। তাই 5-কে তার যথাস্থানে নিয়ে এলাম। তার বাঁ দিকে থাকবে 3, 4। আর ডান দিকে থাকবে 6, 7, 8। এই দুটি অ্যারেকে আবার কুইক সর্ট করতে হবে।

ধাপ 3 : এখন 3, 4 অ্যারের পিভোট হচ্ছে 4। সেটি তার যথাস্থানেই আছে। তার বাঁ দিকে একটি সংখ্যা, তাই সেই অ্যারেও সর্ট করা আছে। আর ডান দিকে কিছু নেই।

ধাপ 4 : এদিকে 6, 7, 8 সংখ্যাগুলোর মধ্যে পিভোট হচ্ছে 8। এটিও তার যথাস্থানে আছে। তাই তার বাঁ দিকের অ্যারেকে (6, 7) আবার কুইক সর্ট করতে হবে।

ধাপ 5 : 6, 7 অ্যারের পিভোট হচ্ছে 7। এটিও তার সঠিক স্থানে আছে। তাই তার বাঁ দিকের অ্যারেকে কুইক সর্ট করতে হবে। সেখানে আছে একটি সংখ্যা (6), তাই সেটি ইতিমধ্যেই সর্ট করা।



ছবি 11.2

আমাদের পুরো অ্যারে কিন্তু সর্ট করা হয়ে গেল।

এখন, পার্টিশন করার পদ্ধতিটি কেমন? আমরা যখন একটি অ্যারের মধ্যে সর্বশেষ উপাদানটি পিভোট হিসেবে নির্বাচন করলাম, তারপরে অ্যারের শুরু থেকে পিভোটের আগ পর্যন্ত এক-এক করে প্রতিটি সংখ্যাকে পিভোটের সঙ্গে তুলনা করে দেখব যে, সেটি পিভোটের চেয়ে ছোটো কি না। এজন্য আমরা একটি লুপ ব্যবহার করব। আর পিভোটের চেয়ে ছোটো সংখ্যাকে তো পিভোটের আগে রাখতে হবে। তাই সেই সংখ্যার খোঁজ রাখার জন্য আমরা একটি ভ্যারিয়েবল i ব্যবহার করব, যার শুরুতে মান হবে `low-1`। এতদূর পর্যন্ত যেটুকু বলা হলো, কোড লিখলে দাঁড়াবে এরকম—

```
int partition(int A[], int low, int high)
{
    int pivot, i, j, t;
```

```
    pivot = A[high];

    for (i = low-1, j=low; j < high; j++) {
        if (A[j] < pivot) {
```

এখন j ইনডেক্সের সংখ্যাটি (অর্থাৎ, A[j]) যদি পিভোটের চেয়ে ছোটো হয়, তাহলে তাকে আমাদেরকে রাখতে হবে i-তম স্থানে। তবে শুরুতে i-এর মান যেহেতু low-1, তাই i-এর মানকে 1 বাড়িয়ে নিতে হবে। তাই i-এর মান এক বাড়ানোর পরে, A[j]-কে আমরা রাখব A[i]-তে। তাহলে A[i]-তে আগে যেটি ছিল, সেটি কোথায় যাবে? সেটিকে আমরা A[j]-তে রেখে দেব। অর্থাৎ, A[i] ও A[j]-কে অদলবদল করব। এজন্য আমরা একটি ভ্যারিয়েবল t-এর সাহায্য নেব।

```
for (i = low-1, j=low; j < high; j++) {
    if (A[j] < pivot) {
        i += 1;
        t = A[j];
        A[j] = A[i];
        A[i] = t;
    }
}
```

এখানে এসেই অনেকের মনে প্রশ্ন আসবে, এরকম করার কারণ কী? A[j] < pivot হলে কেন আমাদেরকে A[j] ও A[i]-এর মধ্যে স্থানের অদলবদল করতে হবে? কারণ A[j] যখন pivot-এর চেয়ে সমান কিংবা বড়ো, তখন কিন্তু আমরা কিছু করছি না, বরং লুপে j-এর মান এক বৃদ্ধি করে পরবর্তী ঘরে চলে যাচ্ছি। তাই j-এর মান বেড়ে চলছে, কিন্তু i থেমে আছে; শেষবার যখন কোনো সংখ্যা A[j] পেয়েছিলাম যেটি pivot-এর চেয়ে ছোটো, সেটিকে যেখানে রাখা হয়েছিল, সেখানে i-এর মান আটকে আছে। আর i-এর মান কেবল তখনই বাড়বে, যখন আমরা একটি A[j] পাব, যেটি pivot-এর চেয়ে ছোটো। এভাবে লুপ চালিয়ে আমরা pivot-এর চেয়ে ছোটো সংখ্যাগুলোকে অ্যারের শুরুর দিকে নিয়ে আসব। আর যেহেতু আমরা A[i] ও A[j] অদলবদল করছি, তাই বড়ো সংখ্যাগুলো অ্যারের পেছনের দিকে চলে যাবে। এখন, লুপটি যখন শেষ হবে, তখন A[i] ও তার আগের সবগুলো সংখ্যা pivot-এর চেয়ে ছোটো। তাই pivot-কে রাখতে হবে A[i+1] ঘরে, আর A[i+1] ঘরে যেই সংখ্যাটি ছিল, তাকে আমরা pivot-এর জায়গায় রেখে দেব। তাহলে পুরো partition ফাংশনটি দাঁড়াবে এরকম—

```
int partition(int A[], int low, int high)
{
    int pivot, i, j, t;
```

```
    pivot = A[high];

    for (i = low-1, j=low; j < high; j++) {
        if (A[j] < pivot) {
            i += 1;

            // A[i] ও A[j]-কে অদলবদল করি
            t = A[j];
            A[j] = A[i];
            A[i] = t;
        }
    }

    // A[i+1] ও A[high]-কে অদলবদল করি
    t = A[high];
    A[high] = A[i+1];
    A[i+1] = t;

    return i+1;
}
```

এই পর্যন্ত এসে কারো কারো কিন্তু বুঝতে কষ্ট হবে। অনেকের মনে হতে পারে যে, কীভাবে কী হলো, ঠিক বুঝলাম না। তখন ওপরের অংশটুকু আরো একবার পড়তে হবে, এবং তারপর নিচে আমি আরেকটি উদাহরণ দিয়েছি, সেটি বোঝার চেষ্টা করতে হবে। হাল ছাড়লে চলবে না।

এখন আমরা দেখি, একটি অ্যারের জন্য `partition` ফাংশন কীভাবে কাজ করে। ধরা যাক, আমাদের অ্যারে হচ্ছে,

`A = [ 6, 3, 8, 4, 7, 5];` এখানে, `low = 0, high = 5`।

তাই, `pivot = A[high] = A[5] = 5`।

এখন, শুরুতে, `i = low - 1  = 0 - 1 = -1`।

এখন আমরা লুপের প্রতিটি ধাপ কীভাবে কাজ করে সেটি দেখব।

ধাপ 1 : লুপের ভেতরে `j`-এর মান শুরু হবে 0 থেকে। প্রথমে পরীক্ষা করা হবে, `A[0] < pivot` শর্তটি সত্য কি না। `A[0]` হচ্ছে 6, তাই শর্তটি মিথ্যা।

ধাপ 2 : `j`-এর মান বেড়ে দাঁড়াবে 1। `A[1] < pivot` সত্য (কারণ `A[1]`-এ আছে 3)। তাই আমরা `if` ব্লকের ভেতরে ঢুকব। প্রথমে i-এর মান এক বাড়াব। তাই `i`-এর মান হবে 0 (−1 এর সঙ্গে 1 যোগ)। তারপর `A[0]` ও `A[1]` অদলবদল করব। সেটি করলে `A[0]` হবে 3, আর `A[1]` হবে 6। অ্যারের অবস্থা দাঁড়াবে `A = [3, 6, 8, 4, 7, 5]`।

ধাপ 3 : লুপের ভেতর `j`-এর মান বেড়ে হবে 2। `A[2] < pivot` মিথ্যা।

ধাপ 4 : `j`-এর মান আরো এক বেড়ে হবে 3। `A[3] < pivot` সত্য। তাই আমরা `i`-এর মান এক বাড়াব। তাহলে `i`-এর মান হবে 1। এই যে `i`-এর মান 1, তার মানে হচ্ছে `A[i]`-তে আমরা `A[j]`-কে রাখতে পারব (কারণ `A[j], pivot`-এর চেয়ে ছোটো)। আর `A[i]`-তে যেই মানটি ছিল, তাকে তো কোথাও রাখা লাগবে, নইলে সে লিস্ট থেকে হারিয়ে যাবে। তাই আমরা `A[j]`-তে `A[i]`-কে রেখে দেব। তাহলে `A[1]`-এ রাখব 4 (অর্থাৎ `A[3]`), আর `A[3]`-তে রাখব `A[1]`-এ যেটি ছিল, অর্থাৎ 6। তাহলে অ্যারের অবস্থা দাঁড়াবে, `A = [3, 4, 8, 6, 7, 5]`।

ধাপ 5 : তারপর `j`-এর মান 1 বেড়ে হবে 4। `A[4]` কিন্তু `pivot`-এর চেয়ে ছোটো নয়। তারপর প্রোগ্রাম লুপ থেকে বের হয়ে যাবে।

লুপ থেকে বের হওয়ার পরে ফাংশনের ভেতরে আরো দুটি স্টেটমেন্ট চালানো বাকি আছে। এখন, `i`-এর সর্বশেষ মান ছিল 1। তার মানে হচ্ছে `A[1]`-এ যেই সংখ্যা আছে, সেটি `pivot`-এর চেয়ে ছোটো। তাই `pivot`-কে রাখতে হবে `A[i+1]` অর্থাৎ `A[2]`-তে এবং `A[2]`-কে রাখব `pivot`-এর জায়গায়, অর্থাৎ `A[high]`-এ। অর্থাৎ 8 ও 5-এর মধ্যে জায়গা অদলবদল হবে। এই কাজটি করার পরে অ্যারের সর্বশেষ অবস্থা এমন, `A = [3, 4, 5, 6, 7, 8]`। তাহলে `pivot` (মানে 5)-এর বর্তমান অবস্থান হচ্ছে `i+1` বা 2। ফাংশনের শেষ লাইনে আমরা 5-এর বর্তমান অবস্থান বা ইনডেক্স, অর্থাৎ `i+1` রিটার্ন করে দেব।

তাহলে আমরা এখন কুইক সর্ট করার জন্য সহজেই একটি প্রোগ্রাম লিখে ফেলতে পারব। `quick_sort` ফাংশনের ভেতরে, শুরুতেই আমরা পরীক্ষা করে নেব যে, `low >= high` কি না। `low > high` এর অর্থ হচ্ছে অ্যারেতে কিছু নেই, আর `low` যদি `high`-এর সমান হয়, তার অর্থ হচ্ছে অ্যারেতে একটিমাত্র উপাদান আছে। উভয় ক্ষেত্রেই আমরা ফাংশন থেকে রিটার্ন করব, কারণ অ্যারেতে একটি বা তার চেয়ে কম উপাদান থাকার মানে হচ্ছে, সেই অ্যারে সর্ট করা আছে, তাকে আর সর্ট করার প্রয়োজন নেই।

```
#include <stdio.h>

int partition(int A[], int low, int high)
{
```

```
    int pivot, i, j, t;

    pivot = A[high];

    for (i = low-1, j=low; j < high; j++) {
        if (A[j] < pivot) {
            i += 1;

            t = A[j];
            A[j] = A[i];
            A[i] = t;
        }
    }

    t = A[high];
    A[high] = A[i+1];
    A[i+1] = t;

    return i+1;
}

void quick_sort(int A[], int low, int high)
{
    if (low >= high) {
        return;
    }
    int p; // p হচ্ছে পিভোট-এর ইনডেক্স।

    // পার্টিশন ফাংশনটি পিভোটের ইনডেক্স রিটার্ন করে
    p = partition(A, low, high);

    // পার্টিশন ফাংশন কল হওয়ার পরে, A[p]-তে যেই সংখ্যাটি আছে, সেটি সঠিক
    // স্থানে আছে। তাই আমরা এর আগের এবং এর পরের সংখ্যাগুলো সর্ট করব।

    // অ্যারের low থেকে p-1 ইনডেক্স পর্যন্ত কুইক সর্ট করি
    quick_sort(A, low, p-1);

    // অ্যারের p+1 থেকে high ইনডেক্স পর্যন্ত কুইক সর্ট করি
    quick_sort(A, p+1, high);
}
```

```
int main()
{
    int i, n = 8;
    int A[] = {1, 5, 6, 3, 5, 8, 7, 2, 9};

    quick_sort(A, 0, n);

    for (i = 0; i <= n; i++) {
        printf("%d  ", A[i]);
    }

    printf("\n");
}
```

আমরা প্রোগ্রামটি রান করলে দেখব যে, A অ্যারেটি সর্ট করা হয়ে গিয়েছে।

মার্জ সর্টের বেলায় মার্জ করার জন্য আমাদেরকে অতিরিক্তি অ্যারে ব্যবহার করতে হয়েছে, কিন্তু কুইক সর্টের বেলায় ইনপুটে দেওয়া অ্যারের বাইরে আর কোনো অ্যারে ব্যবহার করতে হয়নি। তাই এই ধরনের সর্টিং অ্যালগরিদমকে বলা হয় ইন-প্লেস (in-place) সর্টিং অ্যালগরিদম।

কাউন্টিং সর্ট (Counting sort)

বইতে এখন পর্যন্ত আমরা বিভিন্ন রকম সর্টিং অ্যালগরিদমের সঙ্গে পরিচিতি হয়েছি। কিছু অ্যালগরিদমের কমপ্লেক্সিটি $O(n^2)$, কিছু অ্যালগরিদমের কমপ্লেক্সিটি $O(n \log n)$। আমরা জেনেছি কুইক সর্ট অ্যালগরিদমের ওয়ার্স্ট কেস কমপ্লেক্সিটি $O(n^2)$ হওয়া সত্ত্বেও বিভিন্ন ক্ষেত্রে এটি সবচেয়ে বহুল ব্যবহৃত সর্টিং অ্যালগরিদম এবং এটির এভারেজ কেস $O(n \log n)$। আবার হিপ সর্ট ও মার্জ সর্টেও ওয়ার্স্ট কেস $O(n \log n)$ তবে এগুলো কুইক সর্টের মতো বহুল ব্যবহৃত নয়। সাধারণত $O(n \log n)$-এর চেয়ে কম কমপ্লেক্সিটিতে সর্টিং করা যায় না। তবে একটি বিশেষ ক্ষেত্রে এর ব্যতিক্রম রয়েছে। আমাদের যদি mসংখ্যক ধনাত্মক সংখ্যা সর্ট করতে হয়, আর সেগুলো যদি একটি নির্দিষ্ট সীমা n-এর মধ্যে হয়, তাহলে সে ক্ষেত্রে $O(m + n)$ কমপ্লেক্সিটিতে সর্টিং করা সম্ভব। এজন্য একটু চিন্তাভাবনা করতে হবে।

👁 এই সর্টিং অ্যালগরিদমটি ব্যক্তিগতভাবে আমার খুব প্রিয়। এই অ্যালগরিদম 1954 সালে আবিষ্কৃত হয়। তো 2003 সালে একটি জাতীয় পর্যায়ের প্রোগ্রামিং প্রতিযোগিতায় আমি একটি সমস্যার সমাধান করার চেষ্টা করছিলাম। প্রথমে কুইক সর্ট, পরে কুইক সর্ট ও

বাইনারি সার্চ একসঙ্গে ব্যবহার করে সমাধান করার চেষ্টা করি এবং উভয় ক্ষেত্রেই জাজ আমাকে টাইম লিমিট এক্সিডেড (time limit exceeded) এরর দেয়। তারপর ঘণ্টা খানেকের মধ্যেই আমি আরেকভাবে সমাধান করি এবং তখন আমার সমাধান সঠিক বলে বিবেচিত হয়। পরবর্তী সময়ে আমি যখন অ্যালগরিদম বইতে কাউন্টিং সর্ট পড়ি, তখন বুঝতে পারি যে, সেই সময় চিন্তাভাবনা করে বের করা সমাধানটি ছিল কাউন্টিং সর্ট।

কাউন্টিং সর্ট করতে হলে আমাদেরকে নিচের বিষয়গুলো খেয়াল রাখতে হবে :

- ইনপুটে কেবল পূর্ণসংখ্যা থাকতে পারবে এবং কোনো ঋণাত্মক সংখ্যা থাকতে পারবে না।
- ইনপুটে সর্বোচ্চ সংখ্যা কত হতে পারে, সেই সংখ্যাটির মান জানা থাকতে হবে।

এখন সর্বোচ্চ সংখ্যার মান যত, তার চেয়ে 1 বেশি সাইজের একটি অ্যারে তৈরি করা লাগবে। ধরা যাক, সেই অ্যারের নাম হচ্ছে `count`। শুরুতে অ্যারের সবগুলো উপাদান `0` থাকবে। সব ঘরে `0` দেওয়ার মানে হচ্ছে, অ্যারেতে প্রতিটি সংখ্যা `0` বার আছে। যেমন, `count[x]`-এর মান `0` অর্থাৎ `x` এই অ্যারেতে `0` বার আছে। তারপরে আমরা ইনপুটের সংখ্যাগুলো একটি লুপ ব্যবহার করে ট্রাভার্স করব এবং প্রতিটি সংখ্যার জন্য অ্যারের ভেতরে তার জন্য নির্দিষ্ট ঘরে যে সংখ্যাটি আছে, তার মান এক বাড়িয়ে দেব। অর্থাৎ সংখ্যাটি যদি `x` হয়, তাহলে `count[x] = count[x] + 1`। ইনপুটের সবগুলো সংখ্যার জন্য কাজটি করা হয়ে গেলে, `count[x]` এর মান যদি c হয়, তার মানে হচ্ছে, ওই অ্যারেতে x আছে cসংখ্যক বার। তারপরে আমাদের আরেকটি কাজ করতে হবে। তবে সেই কাজটি করার আগে আমরা এতক্ষণ যে বর্ণনা দেওয়া হলো, তার একটি উদাহরণ দেখব।

ধরা যাক, আমাকে বলা হলো যে, এই সংখ্যাগুলো ছোটো থেকে বড়ো ক্রমে সাজাতে হবে : `3, 4, 1, 6, 2, 4, 9, 7, 8, 4, 2, 1`।

```
int A[] = {3, 4, 1, 6, 2, 4, 9, 7, 8, 4, 2, 1};
```

আর এটিও বলে দেওয়া হলো যে, ইনপুটে সর্বোচ্চ সংখ্যা থাকতে পারে 10। তাই আমি 11 সাইজের একটি অ্যারে তৈরি করে ফেলি (তাহলে 10 পর্যন্ত ইনডেক্স ব্যবহার করতে পারব)।

```
int count[11];

for (i = 0; i < 11; i++) {
    count[i] = 0;
}
```

এখন একটি লুপের সাহায্যে A অ্যারের প্রতিটি সংখ্যা x-এর জন্য আমরা `count[x] = count[x] + 1` লিখব। তার মানে, x যেহেতু পেলাম, আগে যতবার x পেয়েছিলাম (অর্থাৎ `count[x]`), তার চেয়ে এখন একবার বেশি পেলাম।

```
for (i = 0; i < n; i++) {
    x = A[i];
    count[x] = count[x] + 1;
}
```

লুপটি যতবার চলবে, ততবার count লিস্টের অবস্থা পরিবর্তিত হবে, নিচের ছবির মতো :

index	count	3	4	1	6	2	4	9	7	8	4	2	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	**1**	1	1	1	1	1	1	1	1	**2**
2	0	0	0	0	0	**1**	1	1	1	1	1	**2**	2
3	0	**1**	1	1	1	1	1	1	1	1	1	1	1
4	0	0	**1**	1	1	1	**2**	2	2	2	**3**	3	3
5	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	**1**	1	1	1	1	1	1	1	1
7	0	0	0	0	0	0	0	0	**1**	1	1	1	1
8	0	0	0	0	0	0	0	0	0	**1**	1	1	1
9	0	0	0	0	0	0	0	**1**	1	1	1	1	1
10	0	0	0	0	0	0	0	0	0	0	0	0	0

লুপ শেষ হওয়ার পরে count অ্যারেটির অবস্থা দেখে নেওয়া যাক।

```
for (i = 0; i < 11; i++) {
    printf("index : %d, value : %d\n", i, count[i]);
}
```

প্রোগ্রামটি রান করলে আমরা আউটপুট পাব এমন—

```
index : 0, value : 0
index : 1, value : 2
index : 2, value : 2
index : 3, value : 1
index : 4, value : 3
index : 5, value : 0
index : 6, value : 1
index : 7, value : 1
index : 8, value : 1
index : 9, value : 1
index : 10, value : 0
```

তাহলে আমরা বলতে পারি, আমাদেরকে সর্ট করার জন্য যেসব সংখ্যা দেওয়া হয়েছিল, সেখানে 0 আছ 0 বার, 1 আছে 2 বার, 2 আছে 2 বার, 3 আছে 1 বার, 4 আছে 3 বার... এরকম আর কি। এখন এই count অ্যারে ব্যবহার করেই কিন্তু আমরা ইনপুটের সংখ্যাগুলোর ছোটো থেকে বড়ো ক্রমে সাজানো অ্যারে তৈরি করতে পারি। 1 আছে 2 বার, তাহলে প্রথম দুটি সংখ্যা হবে, 1, 1। 2 আছে 2 বার। তাহলে পরের দুটি সংখ্যা হবে 2, 2। 3 আছে 1 বার, 4 আছে 3 বার। তাহলে পরবর্তী সংখ্যাগুলো হবে 3, 4, 4, 4। তাহলে এতটুকু এসে আমাদের সর্ট করা অ্যারের অবস্থা দাঁড়াবে এরকম : 1, 1, 2, 2, 3, 4, 4, 4। তারপরে 5 আছে 0 বার, অর্থাৎ 5 নেই। 6, 7, 8, 9 প্রতিটি সংখ্যা 1 বার করে আছে, তাহলে অ্যারেতে যোগ হবে, 6, 7, 8, 9। 10 আছে 0 বার, মানে অ্যারেতে 10 নেই। তাহলে আমাদের সর্ট করা অ্যারেটি দাঁড়াল এ রকম : 1, 1, 2, 2, 3, 4, 4, 4, 6, 7, 8, 9। আর সেই সঙ্গে সর্ট করার কাজটিও শেষ হলো। তোমাদের কাজ হচ্ছে কাউন্টিং সর্ট করার প্রোগ্রামটি পুরোপুরি লিখে ফেলা। আশা করি, লিখতে বেশি সময় লাগবে না।

অধ্যায় ১২ : আরো বেশি শিখতে হলে

বইটি পড়ার সঙ্গে সঙ্গে কিন্তু তোমাদের প্রতিটি ডেটা স্ট্রাকচার ও অ্যালগরিদম নিজে নিজে ইমপ্লিমেন্ট করতে হবে। তোমাদের মধ্যে যারা প্রোগ্রামিং প্রতিযোগিতার ব্যাপারে আগ্রহী, তারা বিভিন্ন ওয়েবসাইটে এসব ডেটা স্ট্রাকচার ও অ্যালগরিদম ব্যবহার করে বিভিন্ন সমস্যার সমাধান করার চেষ্টা করতে পারো। গুগলে খুঁজলেই তোমরা এরকম ওয়েবসাইট ও প্রোগ্রামিং সমস্যার তালিকা পেয়ে যাবে। প্রোগ্রামিং নিয়ে কোনো জিজ্ঞাসা থাকলে প্রশ্ন করতে পারো প্রোগ্রামাবাদ ওয়েবসাইটে (http://programabad.com) কিংবা ফেসবুকের এই গ্রুপে– http://fb.com/groups/programming.school/।

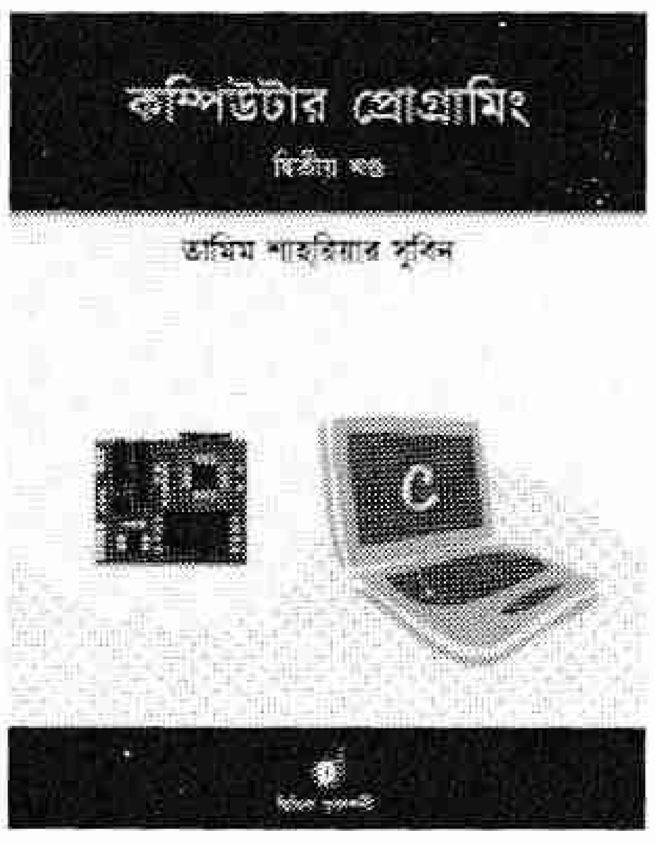
কোন অ্যালগরিদম কীভাবে কাজ করে, সেটি দেখার জন্য তোমরা এই ওয়েবসাইটে যেতে পারো– https://visualgo.net/।

তোমরা যারা আরো বেশি অ্যালগরিদম শিখতে চাও, তোমরা *Introduction to Algorithms* (লেখক : Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest এবং Clifford Stein) বইটি পড়ার চেষ্টা করতে পারো।

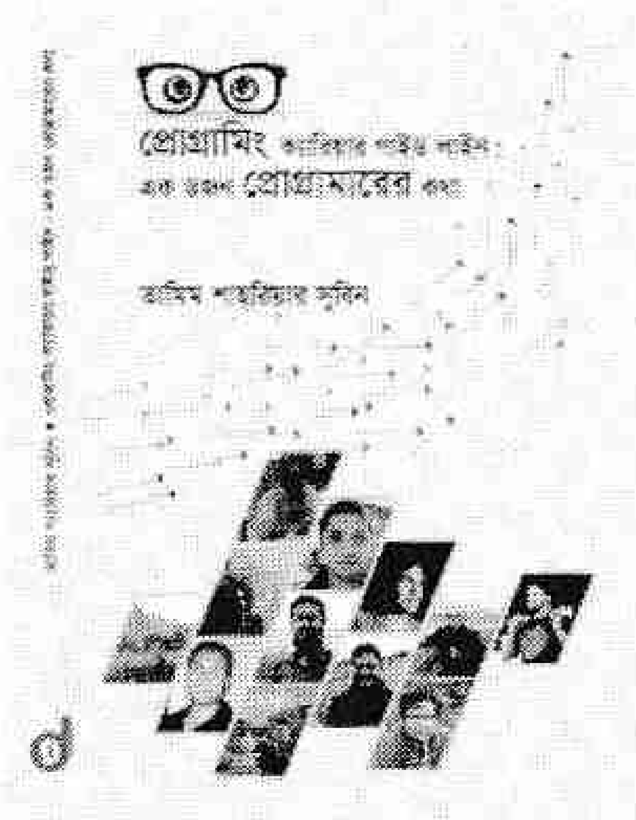
প্রোগ্রামিং প্রতিযোগিতায় আগ্রহীদের জন্য মাহবুবুল হাসানের *প্রোগ্রামিং কনটেস্ট - ডেটা স্ট্রাকচার ও অ্যালগরিদম* বইটি অনেক উপযোগী। আর গ্রাফ অ্যালগরিদম ভালোভাবে শিখতে চাইলে শাফায়েত আশরাফের *গ্রাফ অ্যালগরিদম* বইটি পড়তে পারো।

বই পড়া এবং শত শত ঘণ্টা প্রোগ্রামিং চর্চার মাধ্যমে তোমরা প্রোগ্রামিংয়ের জগতে এগিয়ে যাও। তোমাদের জন্য ভালোবাসা।

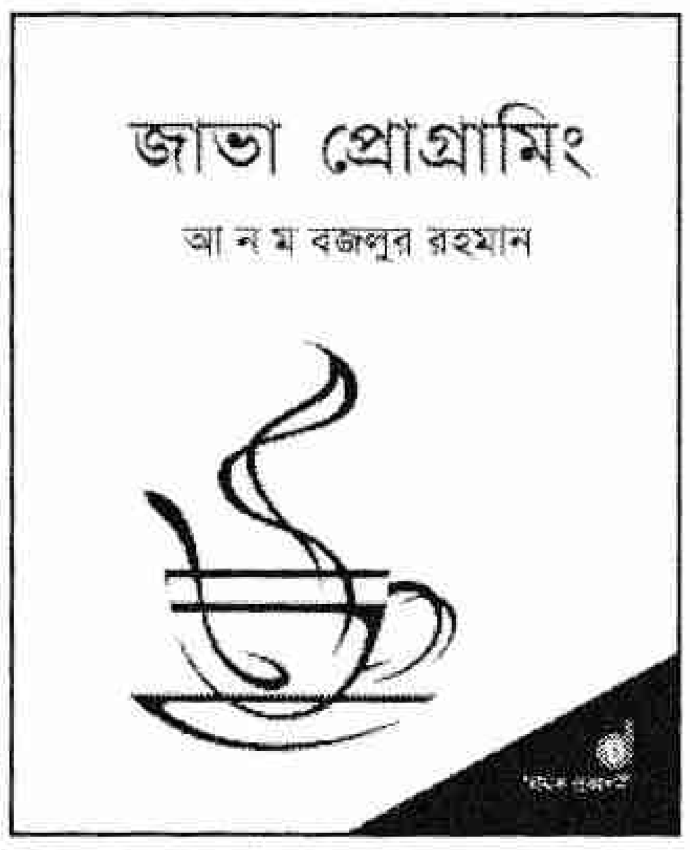
দ্বিমিক প্রকাশনীর আরো বই



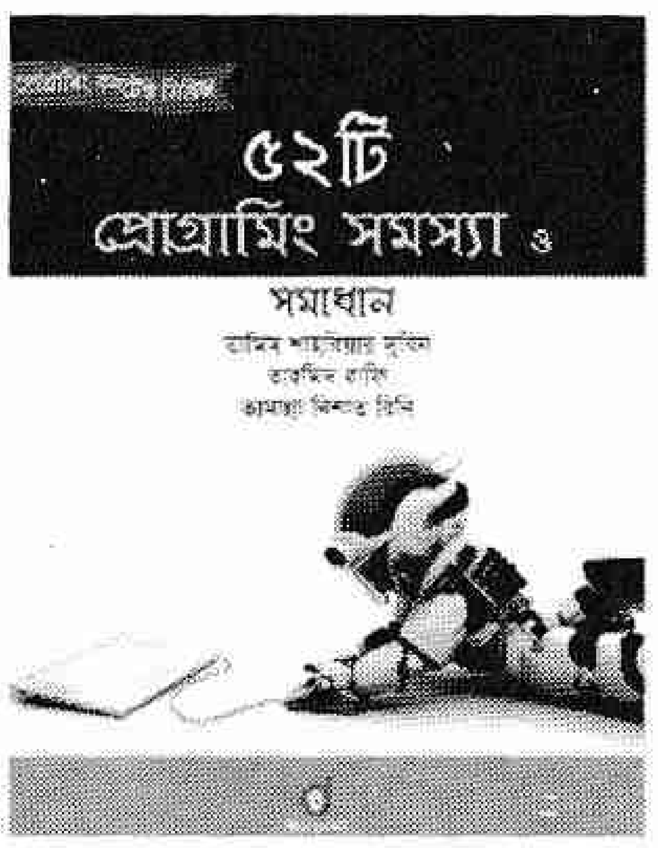
কম্পিউটার প্রোগ্রামিং
২য় খণ্ড
–তামিম শাহ্‌রিয়ার সুবিন



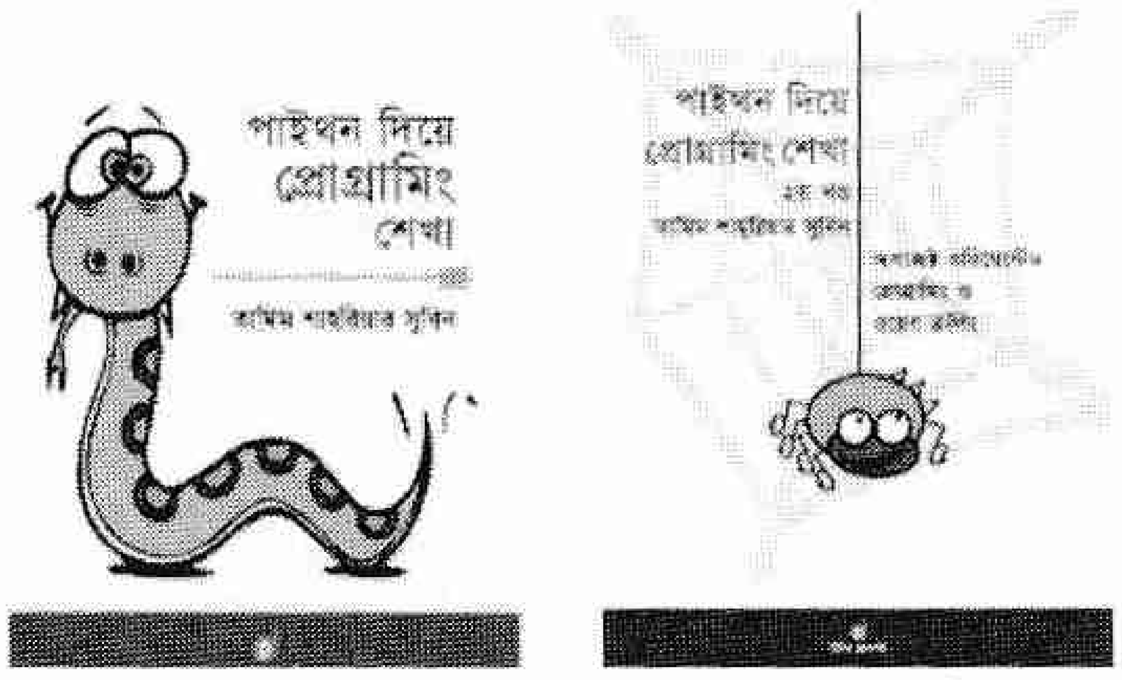
প্রোগ্রামিং ক্যারিয়ার গাইডলাইন : এক ডজন প্রোগ্রামারের কথা
–তামিম শাহ্‌রিয়ার সুবিন



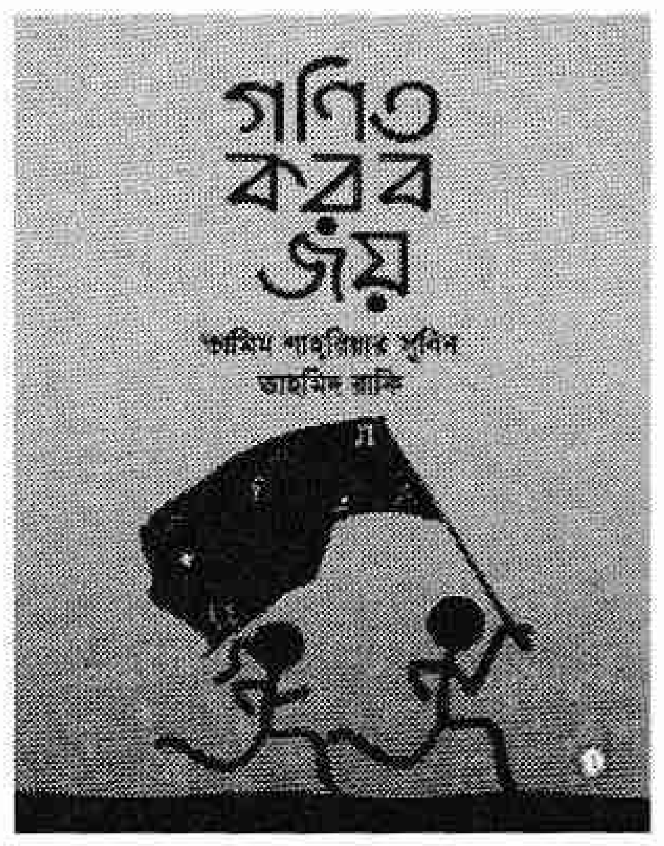
জাভা প্রোগ্রামিং
–আ ন ম বজলুর রহমান



৫২টি প্রোগ্রামিং সমস্যা ও সমাধান
–তামিম শাহ্‌রিয়ার সুবিন
তাহমিদ রাফি
তামান্না নিশাত রিনি



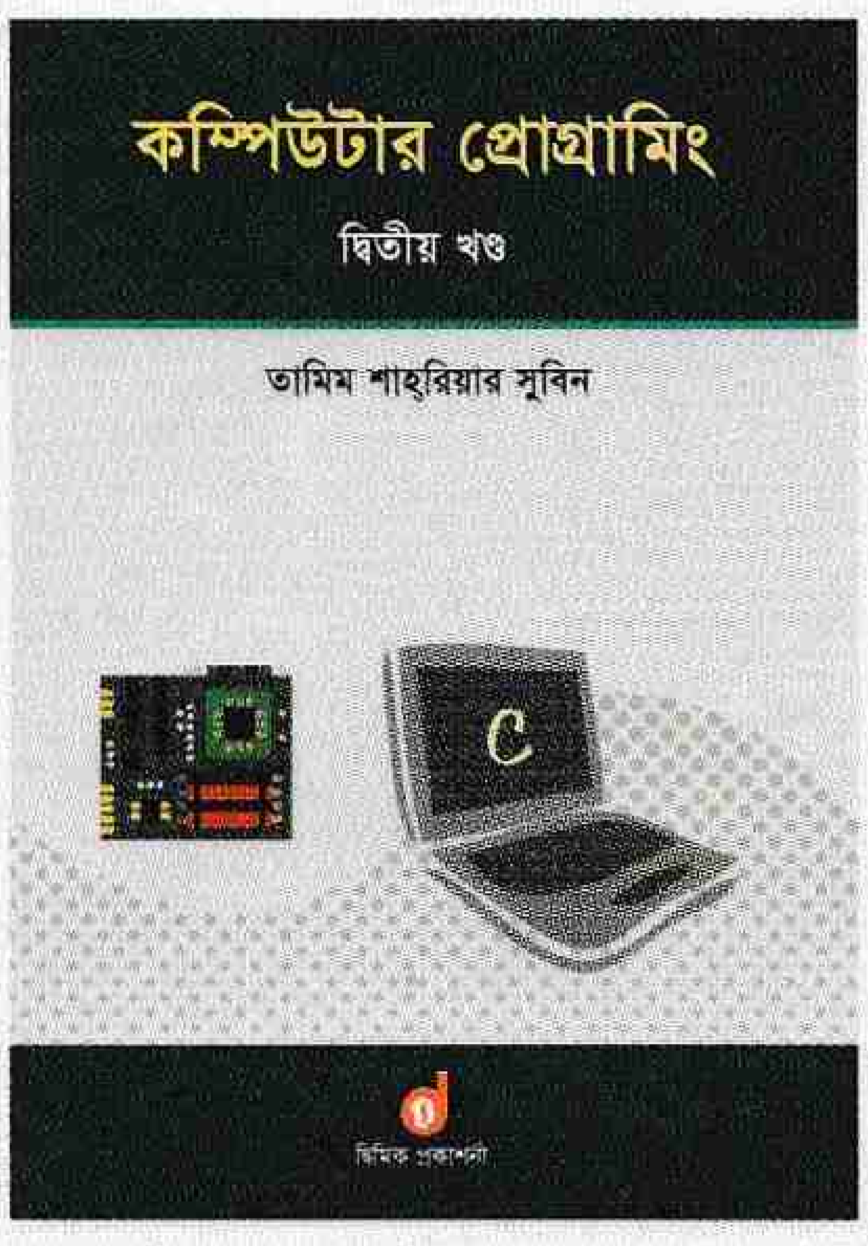
পাইথন দিয়ে প্রোগ্রামিং শেখা ১ম খণ্ড ও ২য় খণ্ড
–তামিম শাহ্‌রিয়ার সুবিন



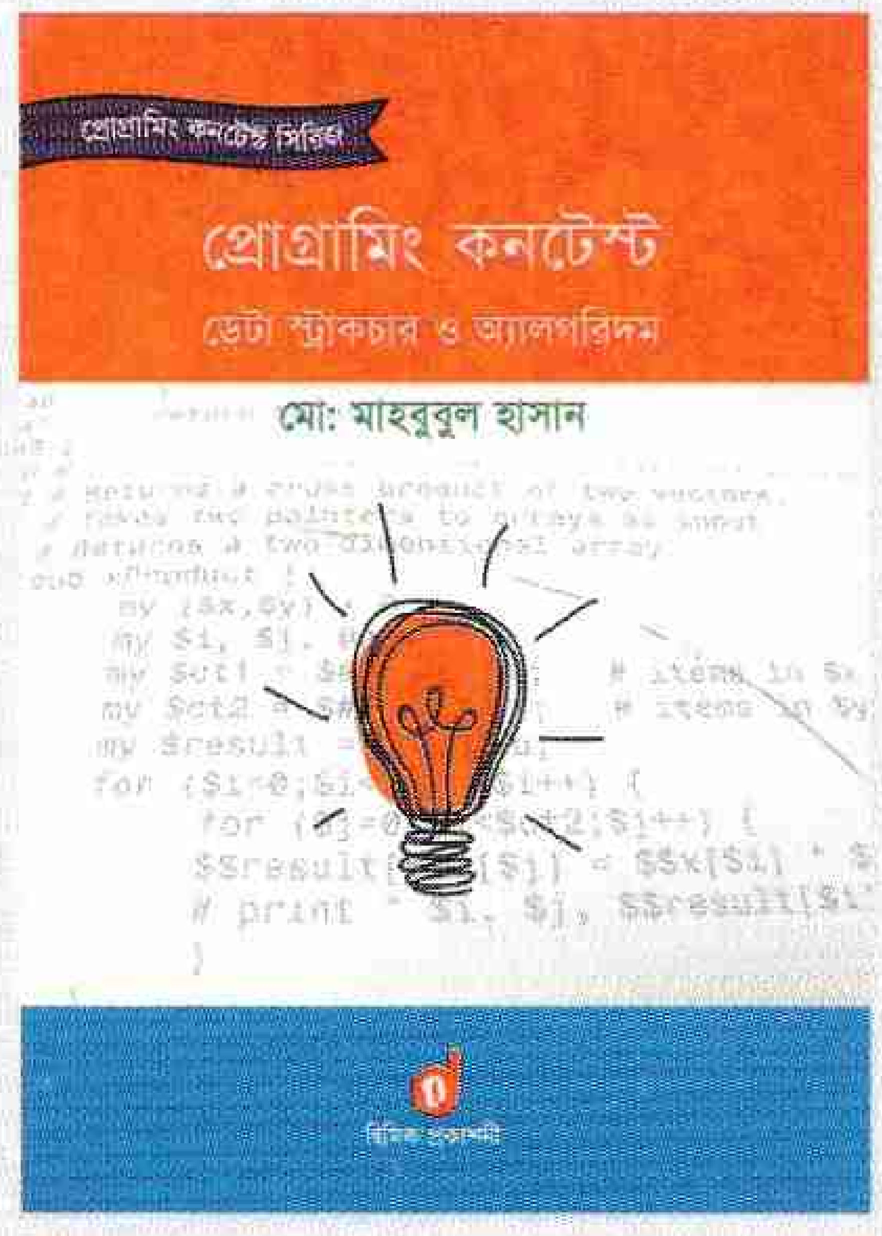
গণিত করব জয়
–তামিম শাহ্‌রিয়ার সুবিন
তাহমিদ রাফি

- প্রোগ্রামিং কনটেস্ট : ডেটা স্ট্রাকচার ও অ্যালগরিদম – মো: মাহবুবুল হাসান
- গ্রাফ অ্যালগরিদম – শাফায়েত আশরাফ
- পাইথন দিয়ে প্রোগ্রামিং শেখা ৩য় খণ্ড – তামিম শাহ্‌রিয়ার সুবিন
- জাভায় থ্রেড প্রোগ্রামিং – আ ন ম বজলুর রহমান
- প্রোগ্রামিংয়ের আশ্চর্য জগৎ – তামিম শাহ্‌রিয়ার সুবিন
- পিএলসি ও ইন্ডাস্ট্রিয়াল অটোমেশন – স্বাগত ভট্টাচার্য্য

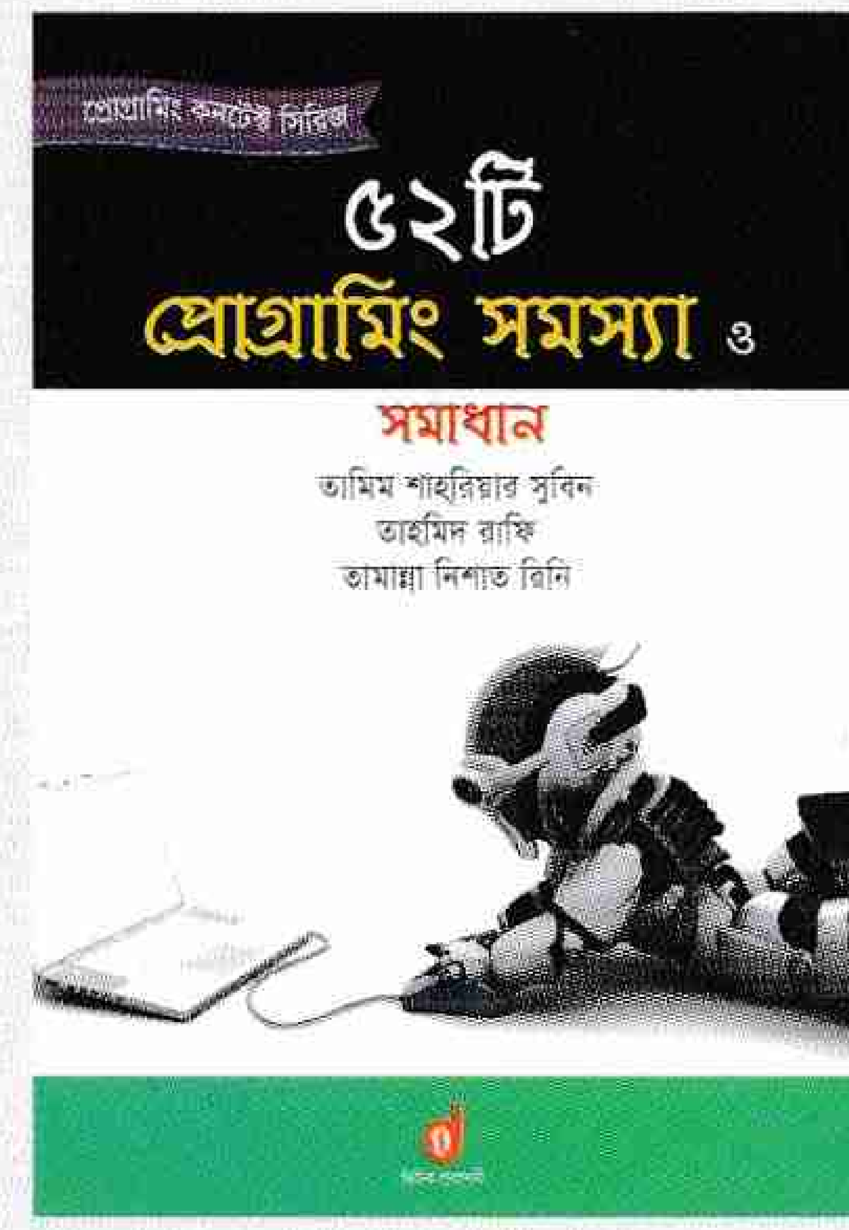
দ্বিমিক প্রকাশনীর অন্যান্য বই



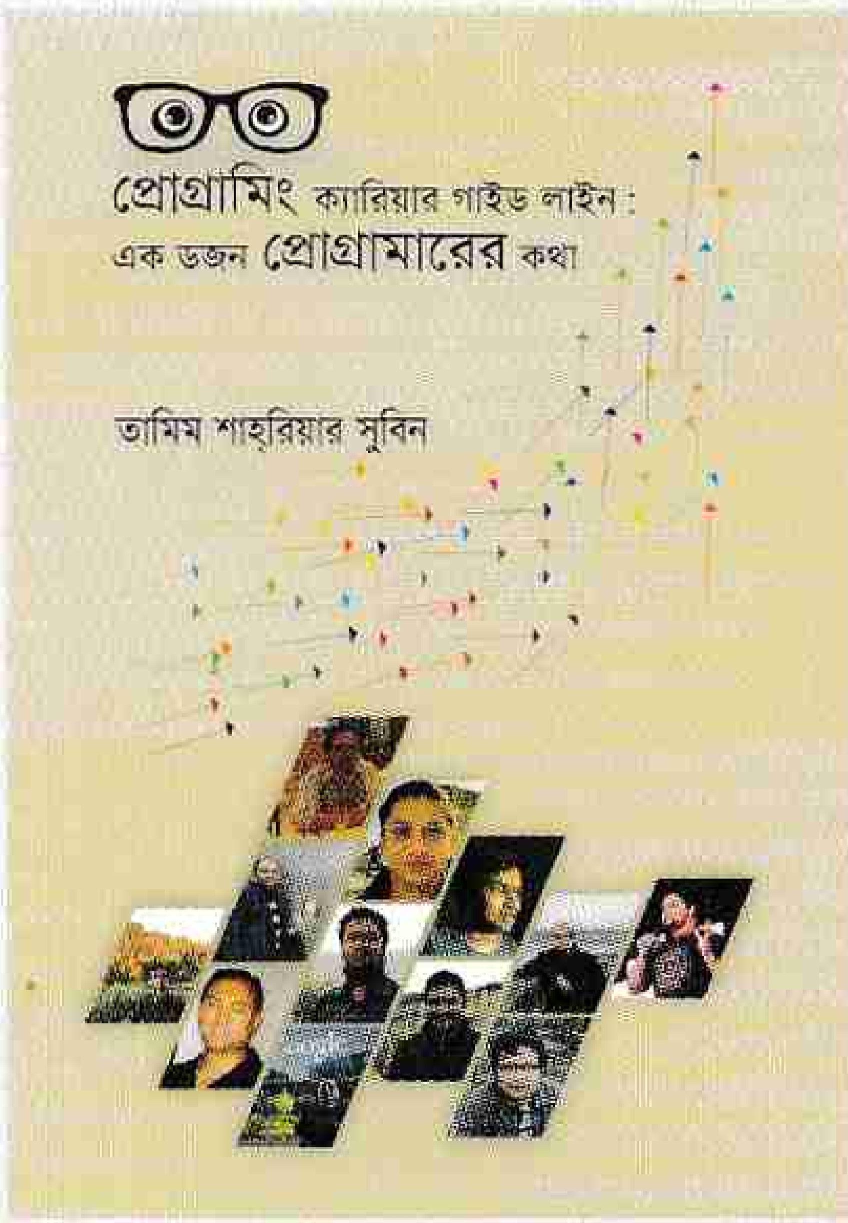
কম্পিউটার প্রোগ্রামিং
দ্বিতীয় খণ্ড
তামিম শাহ্‌রিয়ার সুবিন



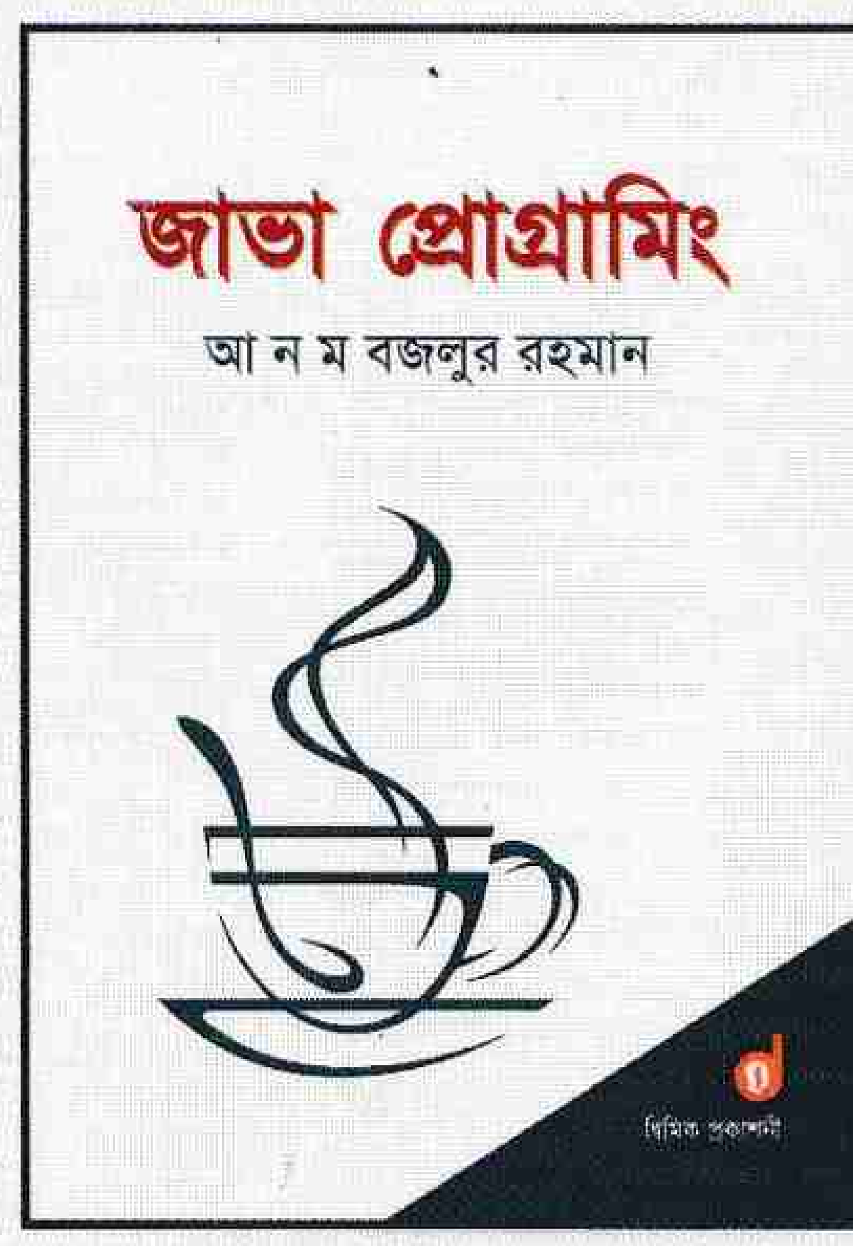
প্রোগ্রামিং কনটেস্ট
ডেটা স্ট্রাকচার ও অ্যালগরিদম
মো: মাহবুবুল হাসান



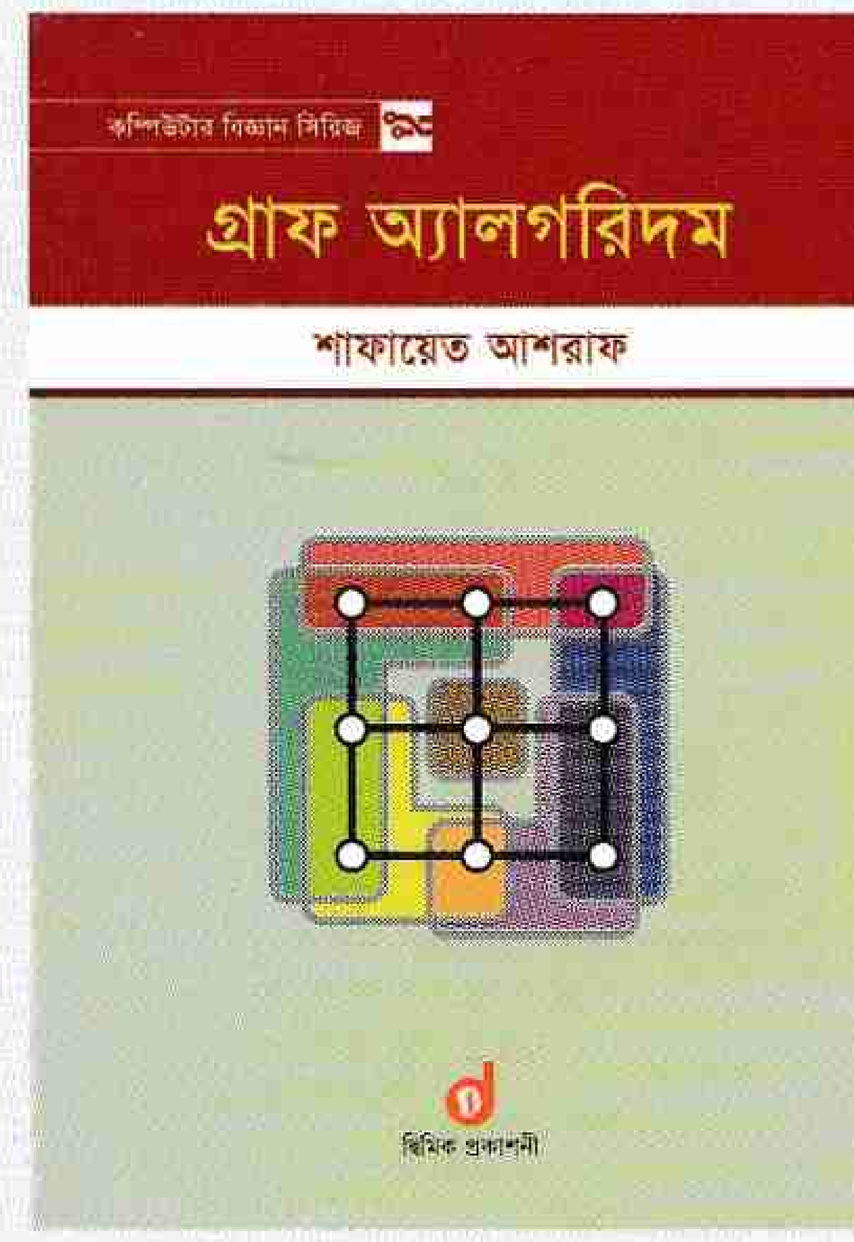
৫২ টি প্রোগ্রামিং সমস্যা ও সমাধান
তামিম শাহ্‌রিয়ার সুবিন
তাহমিদ রাফি
তামান্না নিশাত রিনি



প্রোগ্রামিং ক্যারিয়ার গাইডলাইন
এক ডজন প্রোগ্রামারের কথা
তামিম শাহ্‌রিয়ার সুবিন



জাভা প্রোগ্রামিং
আ ন ম বজলুর রহমান



গ্রাফ অ্যালগরিদম
শাফায়েত আশরাফ

www.dimik.pub
www.dimikprokashoni